UNIVERSIDAD NACIONAL DE SAN LUIS

FACULTAD DE CIENCIAS FÍSICO MATEMÁTICAS Y NATURALES

Estrategias para Interconectar el Dominio del Problema con el Dominio del Programa en Sistemas Multiparadigmas

Tesis

Para optar por el grado de Doctor en Ingeniería Informática

Autor

Enrique Alfredo Miranda

Director: Dr. Daniel Riesco (Universidad Nacional de San Luis)

Co-Director: Dr. Mario Berón (Universidad Nacional de San Luis)

San Luis - Argentina

-Marzo 2018-

A mis queridos viejos,

por su apoyo incondicional.

Para Santi, Juli y Viki,

por su amor y paciencia infinita.

Para el abu Daniel, ingeniero de las cosas simples,

con un doctorado en la vida misma.

Serás lo que hay que ser, si no, eres nada.

José de San Martín

La frase citada es una más resonantes del General José de San Martín y se preserva en el inconsciente de muchas personas como máxima a la que nunca se puede dar por terminada. Sin embargo, considero que para que alguien sea lo que debe ser, su curso de vida se ve alterado por distintos factores que hacen que éste llegue a destino de forma tal que nunca vuelve a ser el mismo. Uno de los factores más influyentes es el humano; es decir, todas aquellas personas que hacen posible que llegues a destino como una persona distinta a aquella que empezó el camino. En esta etapa donde realicé el doctorado mi destino se vio alterado de manera positiva e irreversiblemente por muchas personas a las cuales no puedo dejar de mencionar y plasmar en este documento.

En primer lugar, quiero agradecer a mi familia por ser incondicionales y brindar todo su apoyo y comprensión, mucho más cuando las cosas se pusieron difíciles. A mi familia de siempre: Ruth, "Cabezón", Caro y Richard; también a mi "nueva familia": Viki, Juli y Santi. Quisiera pedirles perdón por haberlos abandonado en varias ocasiones en donde el trabajo requería de mi mayor dedicación y esfuerzo. Ojala pueda compensar ese tiempo con más momentos compartidos con todos ustedes.

Por otro lado quisiera agradecer a esos amigos con los que tuve la oportunidad de compartir la docencia y la investigación: Corina, Edgardo y Hernán. Me brindaron mucho de su tiempo donde vivimos momentos divertidos y trabajamos arduamente con objetivos en común. Siempre estaré agradecido por la dedicación que pusieron cuando los necesité y considero que son excelentes profesionales y mucho mejores compañeros.

Quiero agradecer profundamente a Mario Berón y a Daniel Riesco, no sólo por ser mis directores, sino también por ser grandes colegas y mucho más importante, nobles orientadores. Su dedicación a la investigación y la docencia inspiró en mi la motivación de investigar y perseguir un posgrado de esta magnitud. Quiero que sepan que tienen el mayor de mis respetos y que fue un honor haber trabajado a su lado; ambos son para mi un modelo a seguir desde muchos aspectos.

Resumen

Sin lugar a dudas, una de las tareas más complejas y que más tiempo consume en el ciclo de vida de una aplicación es la de Mantenimiento y Evolución de Software (MES). Dentro del entorno de MES, las tareas que más tiempo demandan son aquellas que debe ejecutar el ingeniero de software para lograr un completo entendimiento del sistema. A partir de la necesidad de asistir al arduo proceso de comprensión mencionado anteriormente, surge una disciplina de la Ingeniería de Software denominada Comprensión de Programas (CP). La CP se presenta como un área de investigación interesante para impulsar el trabajo de MES a través de técnicas y herramientas que asistan al ingeniero de software en la difícil tarea de analizar y comprender sistemas. En esta tesis doctoral se presenta una estrategia de CP que busca asistir al arduo proceso cognitivo que implica la comprensión de sistemas desarrollados usando lenguajes multiparadigma, mediante la interconexión de los Dominios del Problema y del Programa. El primero hace referencia a la salida del sistema en estudio, el segundo se relaciona con los artefactos de software utilizados para producir dicha salida. Esta vinculación es muy importante ya que permite establecer puentes cognitivos que asisten de manera sólida al ingeniero de software en las tareas de comprensión. Para alcanzar este objetivo, se utilizan distintas técnicas de Ingeniería Reversa que permitirán obtener un modelo de caso de uso de UML el cual servirá como medio para lograr la vinculación. Este tipo de modelo es aceptado como un componente provechoso para describir los requisitos de comportamiento para un sistema y como un medio efectivo de comunicación entre los involucrados entorno al mismo. Para derivar este modelo, la estrategia realiza, en términos generales, los siguientes pasos: i) extracción de información estática del sistema relacionada con diferentes tipos de artefactos de software, estableciendo cierto énfasis en los widgets de GUI, los cuales están estrechamente relacionados con el Dominio del Problema del sistema; ii) filtrado de la información extraída; iii) implementación de un proceso de agrupamiento específico que tiene en cuenta la información extraída; y iv) mapeo del modelo de cluster a un modelo de casos de uso de UML. Aunque las actividades desarrolladas en estos pasos se usan con frecuencia en el contexto de Ingeniería Reversa, la estrategia introduce enfoques inusuales con respecto a las propuestas que se encuentran en la literatura dis-

ponible. Más específicamente, la estrategia presenta: un conjunto de métricas que infiere la importancia relativa de un método o función dentro del sistema analizado, un proceso de reducción de información irrelevante y una nueva técnica para agrupar artefactos de software para luego mapear los mismos en un modelo de caso de uso. La evaluación del enfoque sugiere que la estrategia asiste al ingeniero de software a comprender un sistema que proporciona un modelo de caso de uso detallado.

Índice general

Listado de Acrónimos

ARGUI Árbol de Representación de la GUI.

AST *Abstract Sintax Tree*.

BPMN *Business Process Modeling and Notation*.

CDG Conjunción/Disyunción Generalizadas.

CP Comprensión de Programas.

GELMF Grafo Estático de Llamadas a Métodos/Funciones.

GUI *Graphical User Interface*.

IEEE *Institute of Electrical and Electronics Engineers*.

IR Importancia Relativa.

LSP *Logic Scoring of Preference*.

MC Modelos Cognitivos.

MEMC Métodos de Evaluación Multi Criterio.

OMG *Object Management Group*.

SAPSI *Strategy for the Analysis of Program Static Information*.

SUD Grado de Utilidad Estructural o por sus siglas en inglés *Structural Utility Degree*.

TEI Técnicas de Extracción de Información.

UML *Unified Modeling Languaje*.

VS Visualización de Software.

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Lo importante es no dejar de hacerse preguntas.

— Albert Einstein

En las últimas décadas la informática ha demostrado que su campo de aplicación dista mucho de un universo acotado de disciplinas. La misma ha facilitado avances significativos en áreas tan diversas y numerosas de tal forma que intentar listar las mismas de manera exhaustiva sería prácticamente imposible. Cada día se proponen nuevas técnicas, ideas y/o enfoques para solucionar problemas inherentes al mundo actual, muchos de los cuales sólo pueden ser resueltos mediante el uso de sistemas informáticos. Si bien innumerables problemas han sido resueltos, muchos otros siguen aguardando por una solución que incorpore un sistema de información.

En este contexto, el desarrollo de sistemas de información ha crecido de manera acelerada con el paso de los años. Las técnicas y tecnologías utilizadas durante el desarrollo de software han mejorado en proporción a la demanda de sistemas informáticos. Sin embargo, este estrepitoso crecimiento en la industria de desarrollo también ha generado, entre otras cosas, los siguientes consecuentes:

- las empresas de desarrollo están constituidas por diversos recursos humanos y los mismos van cambiando con el paso del tiempo;

- los productos de software requieren modificaciones constantemente por distintos motivos, como por ejemplo, nuevos requerimientos por parte de alguno de los interesados o cambios en el entorno tecnológico.
- los sistemas desarrollados son cada vez más grandes y permiten integrar distintos tipos de tecnologías, por lo tanto, en términos generales son cada vez más complejos;
- debido al ritmo llevado a cabo durante el desarrollo y el constante cambio en los sistemas, es difícil que los mismos estén concisa y completamente documentados.
- muchos productos de software desarrollados con tecnologías o incluso técnicas en desuso se transforman en *sistemas legados* (o también referenciados como *sistemas heredados*) [1]

Dentro del contexto de desarrollo de software, todos estos aspectos han transformado a la etapa de mantenimiento de software una de las más complejas y que más tiempo consume en el ciclo de vida de un sistema (Bennett y Rajlich, 2000; Von Mayrhauser y Vans, 1995; Pigoski, 1996).

1.1. Mantenimiento de Software

El mantenimiento de software es una actividad que engloba muchos aspectos, generalmente, está relacionada con la totalidad del trabajo realizado sobre un sistema de software después que el mismo está en funcionamiento. Esto abarca la corrección de errores; la reestructuración; la mejora en desempeño, usabilidad o cualquier otro atributo de calidad; entre otros. Según la terminología de *Institute of Electrical and Electronics Engineers* (IEEE) ("IEEE Standard Glossary of Software Engineering Terminology", 1990), el mantenimiento de software es:

> *La modificación de un producto software después de su entrega al cliente o usuario para corregir defectos, para mejorar el rendimiento u otras propiedades deseables, o para adaptarlo a un cambio de entorno.*

[1] Un sistema legado se puede describir como cualquier sistema que se resiste de manera significativa a tareas de modificación y evolución del mismo (Bennett y Rajlich, 2000).

La definición anterior señala que el mantenimiento de software es una actividad post-entrega, es decir, se inicia cuando un sistema se entrega al cliente o usuario. Generalmente, está relacionado con la totalidad del trabajo realizado sobre un sistema de software después que el mismo está en funcionamiento ("IEEE Standard Glossary of Software Engineering Terminology", 1990). Esto abarca la corrección de errores; la restructuración; la mejora en desempeño, usabilidad o cualquier otro atributo de calidad; entre otros. Según la IEEE, existen distintos tipos de mantenimiento, de acuerdo al propósito con el que se lleve a cabo dicha actividad: *correctivo, adaptativo, perfectivo y de emergencia* ("IEEE Standard for Software Maintenance", 1993). Todos los tipos de mantenimiento tienen en común lo siguiente: en cada uno se realiza una modificación de un producto de software luego de la entrega del mismo. El mantenimiento *correctivo* se realiza con el objetivo de enmendar errores encontrados; el *adaptativo*, se efectúa con el fin de mantener un programa usable dentro de un ambiente modificado o en modificación; el *perfectivo*, se lleva a cabo para mejorar el desempeño o estabilidad del sistema y el de *emergencia* es un mantenimiento correctivo no planificado con el fin de mantener un sistema operacional.

En general, el costo de mantenimiento de un producto de software a lo largo de toda su vida útil implica más que los costos de su desarrollo. Las empresas destinan cuantiosos recursos en la etapa de mantenimiento en proporción a las distintas etapas del proceso de desarrollo. Con el paso del tiempo diferentes estudios han analizado los porcentajes respecto de los costos en mantenimiento sobre los costos totales, por ejemplo, Lientz (Lientz y Swanson, 1981) menciona que dicho costo representa el 50 % ; McKee (McKee, 1984) y Port (Port, 1988) afirman que el porcentaje oscila entre 60 % y 75 %; Eastwood (Eastwood, 1993) sostiene que porcentaje es del 75 %; mientras que Erlikh (Erlikh, 2000) establece que los costos de mantenimiento representan un 90 %. Si bien los porcentajes difieren de acuerdo a cada investigación y la época en la que se llevaron a cabo los correspondientes estudios, es posible afirmar que la etapa de mantenimiento es sin duda alguna la más costosa en el contexto de desarrollo de software. En este sentido, cualquier enfoque que pueda mejorar la productividad relacionada a las tareas llevadas acabo durante el mantenimiento, claramente impactará directamente en los costos generales del desarrollo y por consiguiente, en la rentabilidad de la empresa de desarrollo.

Algunos autores (Rugaber, 1995; Corbi, 1989; Fyson y Boldyreff, 1998; Murphy y cols., 2006) plantean que entre el 47 % y el 62 % del tiempo de mantenimiento está destinado específicamente a tareas relacionadas con el entendimiento del sistema. Es difícil inferir una razón particular que explique este factor, de hecho esta pérdida de tiempo se podría atribuir a distintos factores como la complejidad inherente de los sistemas de software, la poca experticia por parte de los programadores para llevar a cabo tareas de este tipo, el desconocimiento del Dominio del Problema del sistema, entre tantas otras. Otro factor determinante es que en muchos casos, el proceso de comprensión se lleva a cabo de forma prácticamente manual, donde el desarrollador pierde mucho de su tiempo buscando información que responda sus dudas respecto al sistema (Rugaber, 1995; Ko y cols., 2007; Murphy y cols., 2006). Aún cuando en las últimas tres décadas se han propuesto enfoques y herramientas que asisten a la comprensión, la industria de software demora en adoptarlos. Algunos estudios realizados en contextos industriales han comprobado que los desarrolladores no tienen conocimiento de dichas técnicas y herramientas revelando una brecha entre la industria y la investigación en la disciplina (Ko y cols., 2007; Maalej y cols., 2014)

1.2. Comprensión de Programas

Siempre que se realice un cambio a una pieza de software, es importante que el ingeniero de software obtenga un completo entendimiento de la estructura, comportamiento y funcionamiento de la parte del sistema que se está modificando. Es sobre la base de este entendimiento, que se pueden generar las propuestas de modificación para lograr los objetivos en la etapa de mantenimiento.

Los ingenieros de software encargados del mantenimiento, empeñan mucho de su tiempo leyendo el código y la documentación complementaria para comprender su lógica, propósito y estructura. En este contexto, el entendimiento de un sistema no es tarea fácil, ya que generalmente, la persona que realiza el mantenimiento no es la misma que escribió el código o bien ha pasado un período de tiempo considerable desde que esta persona finalizó las tareas de desarrollo. En este sentido, Eagleson refleja claramente este aspecto en forma de ley:

> *"Cualquier pieza de código que no se haya visto desde hace seis meses o más, bien podría haber sido escrita por alguien más".*

La ley expone una visión de lo difícil que puede resultar mantener un sistema de software, al punto tal que la misma persona que ha desarrollado un sistema particular puede resultarle desconocido luego de un determinado tiempo. Además, aunque la ley refleja el problema referenciado, muchos expertos afirman que no necesariamente deben ser seis meses o más, sino que dicho fenómeno se podría dar a partir de tres semanas. A todo lo anterior se le debe sumar que los sistemas en su gran mayoría no cuentan con documentación que cumpla ciertos requisitos básicos como por ejemplo que la misma se encuentre actualizada, que sea consistente, que esté completa, entre otros.

Es evidente que todos los aspectos mencionados anteriormente han sido los motivos que han impulsado el surgimiento de distintos recursos para asistir al ingeniero de software en la difícil tarea de comprender un sistema. En este contexto, surge una disciplina denominada Comprensión de Programas (CP), la cual se define de la siguiente manera:

> *Una disciplina de la Ingeniería del Software dirigida a proveer modelos, métodos, técnicas y herramientas, basada en un proceso de aprendizaje específico y procesos de ingeniería, a fin de encontrar un conocimiento más profundo acerca de un sistema de software* (Berón, 2010).

La CP surge como un área de investigación que impulsa el trabajo de mantenimiento y evolución del software a través de técnicas y herramientas que ayuden al ingeniero de software en el análisis y la comprensión de los sistemas.

A través de un extenso estudio y análisis de herramientas de comprensión (Rugaber, 1995; Brooks, 1983; Berón, 2010; Von Mayrhauser y Vans, 1995; Storey, 2005), se pudo comprobar que para lograr una comprensión efectiva del sistema bajo estudio, el sujeto cognoscente (llámese, programador, desarrollador, ingeniero de software o *program reader* (Oliveira, 2009)) debe interrelacionar el Dominio del Problema con el Dominio del Programa. El primer dominio, hace referencia a la salida del sistema. El segundo a las componentes de software usadas para producir dicha salida. Claramente, existe una relación (o relaciones) concreta y robusta entre ambos dominios. Por ejemplo, en un sistema de alumnos de una universidad, existe una clara relación entre la funcionalidad de

consultar el promedio para un alumno y los artefactos de software que permiten computar dicha funcionalidad (campos de texto y rótulos en la interfaz gráfica, funciones que implementan el promedio, variables, constantes, entre otros).

Durante la implementación del sistema el desarrollador va construyendo (desde su perspectiva) la relación entre los dominios y de alguna manera la misma se encuentra activa durante durante el ciclo de desarrollo. Sin embargo, en la etapa de mantenimiento y evolución del software es muy posible que dicha relación pierda fuerza y en determinados casos llegue a desaparecer. La reconstrucción de este tipo de relación es compleja y requiere de estrategias de comprensión efectivas que sean elaboradas tomando como base diferentes temáticas como lo son los *Modelos Cognitivos, Técnicas de Extracción de la Información, Administración de la Información* y *Visualización de Software* (Storey, Wong, y Muller, 1997; Storey, 2005; Rugaber, 1995; Berón, 2010; Oliveira, 2009).

1.3. Problema

Actualmente, existen numerosas herramientas de CP con sofisticadas técnicas de exploración de código. La mayoría de estas funcionan de manera adecuada, sin embargo ciertas tareas de comprensión son todavía muy complejas. Muchas herramientas proveen diferentes vistas de la información extraída del sistema como por ejemplo las propuestas por Lanza y su grupo de investigación (Lanza y cols., 2005) o Jerding y Stasko (Jerding y Stasko, 1998). De hecho algunos entornos de desarrollos integrados tienen embebidos este grupo de herramientas para facilitar el desarrollo y reestructuración del sistema. De esta forma el proceso de comprensión de programas se simplifica porque se puede analizar el sistema desde distintos puntos de vista. No obstante, el uso de este tipo de técnicas no asiste al programador en las complejas tareas de abstracción y vinculación de Dominios ya que las vistas provistas brindan sólo un aspecto del código fuente. Una forma de salvar este inconveniente es a través de la elaboración de estrategias que interconecten los Dominios del Problema y Programa. La característica mencionada previamente simplifica la exploración porque el ingeniero de software solamente inspecciona las partes del sistema relacionadas con la funcionalidad de estudio. Este tipo de técnicas proveen información de alto nivel que se puede utilizar para la realización de futuras exploraciones. De esta

manera, se alcanza una considerable reducción de esfuerzos humanos, ergo, se reducen los costos del proyecto, ya que el ingeniero de software se concentra solamente en las actividades de exploración.

En este contexto, a través del estudio del estado del arte de CP es posible inferir los siguientes aspectos:

- El proceso de elaboración de estrategias de vinculación de Dominios es arduo y complejo. El mismo necesita de un estudio profundo de distintas disciplinas tales como *Modelos Cognitivos, Extracción de Información, Administración de la Información, Visualización de Software, Modelos de CP*, entre otras. Por lo tanto, es difícil encontrar numerosas estrategias de vinculación de Dominios que integren todas las disciplinas previamente mencionadas.

- La mayoría de las técnicas y herramientas de CP abordan sólo el Dominio del Programa. Es decir, realizan análisis de los artefactos relativos al código fuente de un programa sin tener en cuenta componentes del Dominio del Problema.

- Ciertos trabajos sólo proponen teorías y técnicas novedosas, pero no presentan herramientas, casos de estudios o Dominios de aplicación interesantes en donde se puedan poner en práctica efectivamente las mismas.

- Algunas técnicas y herramientas de CP que abordan ambos Dominios y además la interconexión entre los mismos, poseen ciertas desventajas que se tornan relevantes en determinados contextos, por ejemplo no proveen vistas provechosas de la información extraída, requieren de una documentación del sistema exhaustiva, no escalan bien para aplicaciones grandes, abordan lenguajes poco utilizados en la industria de software, requieren de documentación y modelos del Dominio del Problema, entre otros.

- Entre las herramientas propuestas no es posible encontrar una que abarque lenguajes multiparadigmas como por ejemplo Python. Este estilo de programación ha tomado cada vez más relevancia ya que es muy utilizado en distintos contextos actuales, como en la industria de desarrollo de software de mediana y gran

escala (Budd y cols., 1995; Vranić, 2002; Wampler y cols., 2010), entornos educacionales (Gayo y cols., 2003; Helminen y Malmi, 2010), entre otros.

Teniendo en cuenta los aspectos destacados en los ítems anteriores, es posible determinar que la mayoría de las herramientas existentes de CP no proponen estrategias robustas y eficaces para interrelacionar los Dominios del Problema y del Programa. Además, del limitado conjunto de herramientas disponibles que abordan dicho enfoque, algunas poseen ciertas desventajas que impide su aplicación a sistemas de gran tamaño escritos en lenguajes ampliamente utilizados en el contexto de desarrollo de software a gran escala.

1.4. Objetivos de la Tesis

En esta sección se presentan los objetivos de la tesis doctoral, considerando en primer lugar el objetivo general o principal y posteriormente los objetivos específicos que permitirán alcanzar el objetivo general.

Objetivo General

Como objetivo general de esta tesis se pretende mejorar la comprensión de sistemas multiparadigmas, por medio de la definición de una estrategia de CP que permita vincular los Dominios del Problema y del Programa de forma cuasi-automática. Como medio para alcanzar dicho objetivo se propone *Strategy for the Analysis of Program Static Information* (SAPSI), una estrategia que asiste al ingeniero de software durante el proceso de comprensión de un sistema.

Objetivos Específicos

En los siguientes apartados se desarrollan de manera concisa los objetivos específicos de este trabajo de tesis doctoral, desagregados a partir del objetivo general previamente mencionado.

Extraer información y construir representaciones del sistema Extraer información estática a partir del código fuente del sistema que permita construir represen-

taciones por medio de las cuales se llevarán a cabo todos los análisis planteados por la estrategia. La estrategia propone analizar los artefactos del código fuente y su vinculación con ciertos componentes estrechamente vinculados con el Dominio del Programa, específicamente los widgets que forman parte de la GUI. La estrategia se propone haciendo uso de una arquitectura en capas front-end/back-end independizando la misma de las construcciones sintácticas de los lenguajes de programación. La capa de los procesos back-end se abstrae de ciertos aspectos relativos a lenguajes y frameworks específicos y realiza el análisis propuesta tomando como base ciertas representaciones construidas por el front-end. Los procesos llevados a cabo por el front-end deben lidiar con las construcciones específicas de cada lenguaje y librerías gráficas utilizadas y construir las representaciones de base que recibe como entrada el back-end.

Definir una métrica para estimar la importancia relativa Definir una métrica que permita estimar la importancia relativa de cada artefacto de software considerado para el análisis respecto de la lógica subyacente del sistema. Dicha métrica debe tener en cuenta ciertos aspectos inherentes a cada elemento "medido", como por ejemplo las invocaciones entre los artefactos o la relación de cada artefacto con la *Graphical User Interface* (GUI) del sistema. Los valores asociados a la medición de todos estos aspectos se deben combinar para obtener un valor que indique la importancia del artefacto de software. Las características antes mencionadas se ajustan a la definición de ciertos tipos de métodos multicriterio mediante los cuales es posible obtener un único valor tomando como base las diferentes mediciones de dichas características. La métrica previamente referenciada debe facilitar el proceso de filtrado de informacion irrelevante y asistir al proceso de clustering.

Definir un algoritmo de filtrado de información Definir un algoritmo de reducción de información basado en la métrica antes descripta que permita filtrar aquellas entidades en la representación construida que no son esenciales para la lógica subyacente del sistema.

Definir una nueva técnica de clustering de software Definir una nueva técnica de clustering de software orientada a la CP y más específicamente a la obtención de

un modelo de casos de uso *Unified Modeling Languaje* (UML) para el sistema analizado.

Obtener un modelo que facilite la vinculación entre Dominios Obtener de forma cuasi-automática un modelo que facilite la vinculación entre el Dominio del Problema y el Dominio del Programa haciendo uso de la estrategia presentada. Para esto se deben tener en cuenta aquellos modelos ampliamente utilizados en el proceso de desarrollo de software.

Desarrollo de una herramienta que implemente la estrategia Debido a que la estrategia propuesta está compuesta por una serie de etapas que pueden ser automatizables en gran medida, se propone desarrollar una herramienta que implemente todos los procesos abarcados por dicha estrategia.

Definir una metodología de evaluación Definir una metodología para evaluar el modelo obtenido por la estrategia en comparación con un modelo ideal estándar (o *"gold standard"*).

1.5. Organización

El resto del informe está organizado de la siguiente manera:

Capítulo 2: *Comprensión de Programas.*

Aproxima una conceptualización de la disciplina Comprensión de Programas y presenta los distintos modelos de comprensión a lo largo de los años y específicamente el modelo en que está basada la propuesta de este trabajo.

Capítulo 3: *Estado del Arte.*

Proporciona un estado del arte que abarca las distintas aristas comprendidas en la investigación.

Capítulo 4: *SAPSI: una estrategia para asistir en la Comprensión de Programas.*

Introduce la estrategia propuesta, se explican conceptos básicos del enfoque y los principales objetivos que el mismo plantea.

Capítulo 5: *Extracción de Información.*

Presenta las técnicas de extracción utilizadas por SAPSI, el tipo de información que se extrae, las estructuras de base usadas para representar la información extraída y los distintos conceptos asociados a esta temática.

Capítulo 6: *Filtrado de Información.*

Define una métrica denominada Importancia Relativa (IR), la cual permite inferir la importancia de un artefacto de software en relación a la lógica subyacente del sistema. Para esto presenta el método de evaluación multicriterio seleccionado para computar dicha métrica y desarrolla los conceptos relacionados con el mismo. Finalmente, propone una técnica de filtrado de información basada en la métrica antes mencionada.

Capítulo 7: *Técnica de Clustering y Generación del Modelo de Casos de Uso.*

Desarrolla la técnica de clustering usada por SAPSI para agrupar artefactos en la representación base y posteriormente expone el método para transformar este modelo de clusters a un modelo de casos de uso para el sistema bajo estudio.

Capítulo 8: *Dupin: una herramienta para inspección de programas.*

Presenta una herramienta que implementa la estrategia propuesta describiendo las principales componentes de su arquitectura y desarrollando las distintas características que la misma presenta.

Capítulo 9: *Evaluación de la Propuesta.*

Introduce una metodología para evaluar los resultados obtenidos con SAPSI y presenta dos casos de estudio para mostrar la aplicabilidad del enfoque.

Capítulo 10: *Conclusiones y Trabajos Futuros.*

Presenta las conclusiones de la tesis doctoral y describe las posibles extensiones de los temas desarrollados. El propósito principal es motivar al lector a investigar algunos de estos tópicos.

Anexo B : *Análisis de la Aplicación Libreta de Contactos.* Presenta toda la información extraída, administrada y representada por la estrategia para el análisis completo

de una libreta de contactos simple desarrollada en lenguaje Python y la librería gráfica GTK.

Anexo B: *Funciones de Conjunción-Disyunción Generalizada.*

Muestra una tabla extendida con todos los operadores de Conjunción/Disyunción Generalizada de LSP.

Anexo C: *Modelos de Casos de Uso para los Sistemas Analizados.*

Anexo D: *Código Fuente y Archivo Glade del Ejemplo Libreta de Direcciones.*

Exhibe los modelos de casos de uso ideales y obtenidos por SAPSI para los sistemas presentados como casos de estudio.

1.6. Publicaciones Derivadas de este Trabajo

- *Using reverse engineering techniques to infer a system use case model* (Enrique Miranda, Mario Berón, Germán Montejano and Daniel Riesco). **Journal of Software: Evolution and Process**. John Wiley & Sons, Ltd. ISSN: 2047-7481. 2017. En revisión (Indexada SCI, IF JCR2016=0.698).

- *Measuring the understandability of WSDL specifications, Web Service Understanding Degree Approach and System* (Mario Marcelo Berón, Hernán Bernardis, Enrique Alfredo Miranda, Daniel Edgardo Riesco, Maria João, Henriques Pedro). **Journal of Computer Science and Information Systems**. ComSIS Comsortium. 2016 ISSN 1820-0214. 13(3). Pag. 779-807. (Indexada SCI, IF JCR2016=0.837).

- *Inferring Use-cases from GUI Analysis* (Enrique Miranda, Corina Abdelahad, Berón Mario, Riesco Daniel) **Journal IEEE Latin America Transactions**. IEEE. 2016. ISSN 1548-0992. 13(12). Pag. 3942-3952. (Index SCI, IF JCR2015=0.44).

- *A Strategy for Detecting and Clustering Functionalities in Object Oriented Systems* (Enrique Miranda, Mario Berón, Daniel Riesco, Germán Montejano, Narayan Debnath). **30th International Conference on Computers and Their Appli-**

cations CATA-2015. Iowa State University. Estados Unidos de América, Ames, Iowa. 2015.

- *WSDLUD: A Metric to Measure the Understanding Degree of WSDL Descriptions.* (Mario Marcelo Berón, Hernán Bernardis, Enrique Alfredo Miranda, Daniel Edgardo Riesco, Maria João Varanda Pereira, and Pedro Rangel Henriques). **Communications in Computer and Information Science. Languages, Applications and Technologies**. Springer, Cham. 2015. ISBN/ISSN: 978-3-319-27653-3. vol 563. Pag. 91-100.

- *Extracción y Análisis de Información Estática Orientada a la Comprensión de Programas para Sistemas OO* (Enrique Miranda, Mario Berón, Daniel Riesco). **Revista de Ciencia y Tecnología**. Facultad de Ingeniería, Universidad de Palermo. 2014. ISSN 1850-0870. EISSN 2344-9217.1(14). Pag. 163 - 181. Revista Ciencia y Tecnología. Universidad de Palermo. (Latin Index).

- *Una Estrategia Orientada a la Comprensión de Programas para Extracción y Análisis de Información Estática en Sistemas OO* (Enrique Miranda, Mario Berón, Daniel Riesco). **II Congreso Nacional de Ingeniería Informática / Sistemas de Información (CONAIISI 2014)**. Páginas 1052-1062. ISSN: 2346-9927. Universidad Nacional de San Luis. San Luis. Argentina. 2015.

Capítulo 2

Comprensión de Programas

Cualquier tonto puede saber. El punto es comprender.

— Albert Einstein

Cuando se habla de comprensión, básicamente se hace referencia a la acción de comprender y a la facultad, capacidad o perspicacia para entender y penetrar las cosas. Comprender el funcionamiento de un sistema está más en relación con el pensamiento sistémico. Básicamente, el pensamiento sistémico es la actividad realizada por la mente con el fin de comprender un sistema y resolver el problema que presenten sus propiedades emergentes; es un modo de pensamiento que contempla el todo y sus partes, así como las conexiones entre éstas. Mientras que un programa es esencialmente un conjunto de instrucciones que una computadora puede interpretar y ejecutar. El mismo puede llevar a cabo una o varias tareas a medida que se va ejecutando. Teniendo en cuenta las definiciones anteriores, se puede decir intuitivamente que, la Comprensión de Programas (CP) consiste en la habilidad de entender varios *componentes*[1] de un programa que componen una aplicación informática y las conexiones que existen entre ellos. Algunos autores han referenciado a esta disciplina usando otros términos como *Program Understanding* (Biggerstaff y cols., 1993; Belmonte y Dugerdil, 2010; Storey y Muller, 1995; Corbi, 1989).

[1]En este contexto, la palabra *componentes* es usada como una referencia en general a las partes que componen un sistema, como por ejemplo módulos, subsistemas, funciones, etc.

Si bien es posible encontrar distintas definiciones de CP en la literatura, una de las más esenciales es la que provee el grupo PCVIA (PCVIA, 2016):

> *Una disciplina de la Ingeniería del Software dirigida a proveer modelos, métodos, técnicas y herramientas, basada en un* ***proceso de aprendizaje*** *específico y* ***procesos de ingeniería****, a fin de encontrar un conocimiento más profundo acerca de un sistema de software. Donde,*
>
> - ***Proceso de aprendizaje:*** *implica el estudio de las Ciencias Cognitivas y la relación de sus principales conceptos con la Ingeniería de Software.*
> - ***Proceso de ingeniería:*** *incluye el estudio de áreas tales como: Visualización de Software, Extracción de la Información, Administración de la Información y Estrategias de Interconexión de Dominios con la finalidad de representar la información del sistema de una manera que enfatice sus principales aspectos.*

Como se mencionó en el capítulo anterior, la CP surge como un área de investigación útil e interesante para impulsar el trabajo de mantenimiento y evolución del software a través de técnicas y herramientas que ayuden al ingeniero de software en el análisis y la comprensión de aplicaciones de computadoras.

2.1. Conceptos Claves

Como otras áreas de investigación, en el contexto de CP también se emplean frases o términos específicos que permiten establecer un marco de entendimiento dentro de la temática. Si bien algunos ya han sido utilizados en el capítulo previo, es importante explicar los mismos con más detalle y de esta manera facilitar la lectura del informe.

Ingeniero de Software

En el contexto de CP, se han utilizado distintos términos para hacer referencia a la persona que está frente a un sistema realizando cualquier tipo de tarea que conlleve el entendimiento del mismo. Ciertos términos han sido empleados con más frecuencia

como *programador* o *mantenedor*, sin embargo los mismos están asociados a actividades específicas. El término *programador* siempre está vinculado con la persona que desarrolla el sistema y que en ciertas ocasiones debe comprender el mismo. El término *mantenedor* está relacionado con la persona que realiza las tareas específicas durante la etapa de mantenimiento del sistema. Otros autores proponen el término *lector del programa* (*program reader*) (Oliveira, 2009), sin embargo dicho término tiene asociado cierta pasividad teniendo en cuenta las tareas que debe llevar a cabo una persona para comprender un sistema.

En otras investigaciones, al igual que en este trabajo doctoral, se utiliza el término *ingeniero de software*. El motivo de la adherencia al uso del mismo se debe a que las tareas implicadas en CP conllevan tener un conocimiento sólido en diversas disciplinas de la Ingeniería de Software.

Más allá del término utilizado por los distintos autores y los basales que se han aplicado para justificar el uso del mismo, es importante que el lector tenga en cuenta el rol de la persona que llevará a cabo las actividades de CP, más allá del término empelado para denominar la misma.

Artefacto de Software

El término artefacto de software ha sido utilizado en el contexto de desarrollo de sistemas desde hace décadas. En este sentido, un artefacto es una pieza discreta de información que es utilizada o producida por un proceso de desarrollo o un sistema existente (Booch y cols., 2004). Por ende, en el contexto de CP se puede concebir al término *artefacto de software* como un componente de un sistema de software desarrollado para un fin determinado, en especial el que no constituye un sistema o subsistema en sí o dispositivo definido. Teniendo en cuenta esta última definición, podría considerarse artefacto de software a cualquier componente extraído en la inspección del sistema. Por ejemplo, componentes de la interfaz gráfica, las funciones de un sistema, modelos (o componentes de los mismos) de un sistema, etc.

Dominio del Problema

Dominio del Problema es el término utilizado para referirse al área de conocimiento o actividad que se caracteriza por un conjunto de conceptos y una terminología que entienden los profesionales de dicha área. En otras palabras, son los conocimientos y recursos de información referentes al entorno para la cual ha sido desarrollado el sistema bajo estudio (Booch y cols., 2004).

En el contexto de CP, es también conocido como el dominio de conocimiento el cual "está relacionado con el resultado final producido y el impacto del mismo con respecto al problema que el sistema resuelve" (Berón, 2010). Normalmente, es considerado como un conjunto de conceptos y relaciones que están fuertemente asociados con actividades y percepciones humanas. Por lo general, dichos conceptos son extraídos recurriendo a análisis de dominio y ontologías (Arango, 1989; Lee y cols., 2006).

Dominio del Programa

Dominio del Programa es el término empleado para referirse a aquellos componentes inherentes al sistemas que integran la implementación del mismo. En otras palabras, se refiere a aquellos artefactos de software estrechamente relacionados con el código fuente del sistema y aquellos conceptos de bajo nivel asociados al mismo. Como ejemplos de los mismos se puede mencionar a ciertos elementos del código fuente como funciones, llamadas a funciones, sentencias de iteración, variables, etc. También es posible identificar otro tipos de artefactos como modelos UML del código fuente, bases de datos, componentes de la interfaz gráfica de usuario, entre otros.

Claramente, este dominio está estrechamente relacionado con aquellos factores tecnológicos inherentes al lenguaje de programación y cómo el programa es ejecutado para producir una salida (Berón, 2010).

Modelo y Diagrama

En el contexto de Ingeniería de Software, muchos autores usan los términos *modelo* y *diagrama* de manera indistinta. En este sentido, el término más utilizado en las publicaciones es *diagrama*. Sin embargo, el *Object Management Group (OMG)* (OMG, 2017),

a través de sus distintos documentos, ha establecido una diferencia sustancial entre los mismos.

Según UML, el término *diagrama* hace referencia a una representación gráfica de un conjunto de elementos, representando la mayoría de las veces como un grafo conexo de nodos (elementos) y arcos (relaciones) (Booch y cols., 2004). Es decir, se refiere a los elementos específicos con los que se construye un modelo en particular. Por otra parte, un *modelo* es una simplificación de la realidad, creada para comprender mejor el sistema que se está creando (o comprendiendo); abstracción semánticamente cerrada de un sistema. Es decir, brinda información acerca de un aspecto del sistema real con el objetivo de facilitar el entendimiento de dicho aspecto (OMG, 2017).

Para ejemplificar el uso de ambos términos es posible destacar los casos de uso de UML. El *modelo* de casos de uso permite conocer las funcionalidades y sus relaciones para un sistema particular (Booch y cols., 2004), mientras que el *diagrama* de casos de uso es la "herramienta" que permite construir el *modelo* de casos de uso. En este sentido, el *diagrama* de casos de uso está compuesto por *casos de uso*, *actores*, *relaciones asociación*, *relaciones* << *include* >>, *relaciones* << *extend* >> y *generalizaciones*.

Es importante remarcar la diferencia entre los términos *modelo* y *diagrama* ya que en ciertas partes del informe se hará referencia a los mismos usando los significados asociados en este apartado.

2.2. Temáticas Estrechamente Relacionadas a la Comprensión de Programas

Para lograr un entendimiento general acerca de la CP, se deben tener en cuenta un conjunto de temáticas relacionadas directamente con la misma; entre las más importantes se encuentran: Modelos Cognitivos, Extracción de la Información, Administración de la Información y Visualización de Software. En los siguientes apartados se describen cada una de las mismas.

2.2.1. Modelos Cognitivos

Un Modelos Cognitivos (MC) es un modelo de funcionamiento mental (Storey, 1998; Brooks, 1983; Von Mayrhauser y Vans, 1995). Los MCs explican los procesos mentales implicados en la generación del conocimiento. Estos permiten desarrollar hipótesis sobre la conformación de estructuras y procesos que acontecen durante el abordaje del objeto a conocer, por parte del sujeto cognoscente. En el caso de la CP, el objeto a conocer es el sistema en estudio, mientras que el sujeto cognoscente es quien intenta comprender el sistema. Por lo tanto, los MCs son importantes durante el proceso de comprensión, ya que permiten entender los diferentes procesos mentales usados por los sujetos para comprender los sistemas.

Dentro del contexto de CP, el análisis de los MCs resulta provechoso ya que los mismos permiten conocer cómo el programador entiende programas. Esta característica es importante debido a que proporciona guías útiles para la elaboración de estrategias de comprensión que asistan al programador en el proceso de comprensión. Las teorías de MC describen los conceptos empleados en este proceso y caracterizan el camino seguido por el programador para elaborar su conocimiento. Actualmente, existe un acuerdo en las descripciones de las componentes del programa como se podrá ver a través de este apartado. Sin embargo, es posible encontrar muchas diferencias en las estrategias empleadas para relacionarlas y construir nuevos conceptos.

Storey y su grupo de investigación en (Storey, 2005) expresan que un MC está compuesto por tres componentes principales:

Conocimiento: se distinguen dos clases de conocimiento:

- **Interno:** se refiere a los conceptos que posee el programador en su estructura de conocimiento.
- **Externo:** describe los conceptos utilizados en el sistema.

Proceso de Asimilación: especifica la estrategia de aprendizaje usada para comprender el sistema. Más adelante en esta sección se describen brevemente diferentes procesos de asimilación.

Modelo Mental: es una representación interna del sistema. Está compuesto por elementos Estáticos y Dinámicos. Para profundizar los conceptos relacionados con esta temática, se puede consultar el trabajo de Storey y su grupo de investigación (Storey, 2005).

En el ámbito de los MCs, muchos expertos en el tema han realizado estudios con el objetivo de observar cómo los programadores entienden los programas (Brooks, 1983; Parnin y Rugaber, 2012; Shneiderman, 1980; Storey, 2005). Como resultado de dichos estudios se han propuesto diferentes MCs que describen los *Procesos de Asimilación* utilizados por los programadores para comprender los sistemas. A continuación se mencionan los más relevantes.

Bottom Up: propone que los programas son entendidos de manera *Bottom Up*, es decir desde conceptos específicos hacia conceptos generales (Shneiderman, 1980). La comprensión *Bottom Up* implica la lectura del programa, luego la construcción de abstracciones y la incorporación de semántica a las mismas. Este proceso se repite hasta obtener un nivel de abstracción que permita entender el programa.

Top Down: sugiere que los programas son entendidos de manera *Top Down*, es decir desde conceptos generales hacia conceptos específicos (Brooks, 1983). La comprensión *Top Down* usa un proceso inverso a la comprensión *Bottom Up*. Esta técnica supone que el programador conoce algo acerca del dominio de la aplicación o la funcionalidad del programa. El programador formula hipótesis y luego lee el código para verificar esas hipótesis; en otras palabras comienza con un nivel de abstracción alto y luego construye niveles más bajos que le permiten verificar las hipótesis.

Knowledge-based: plantea que los programadores son procesadores oportunistas capaces de "explotar señales", ya sea de manera *Bottom Up* o *Top Down* (Letovsky, 1986). Esta teoría posee tres componentes: i) una base de conocimiento, que codifica la aplicación del programador y su experiencia; ii) un modelo mental, que representa el entendimiento del programa por parte del programador; y iii) un proceso de asimilación, que describe cómo evoluciona el modelo mental usando la base de conocimiento del programador y la información del programa.

Systematic and as-needed: expone que los programadores usan uno de los siguientes enfoques (Littman y cols., 1987): i) sistemático (*Systematic*), leyendo el código en detalle y siguiendo el hilo del programa a través de los flujos de control y de datos; o ii) según necesite (*as-needed*), enfocándose solo en el código relacionado con la tarea a realizar.

Integrated approaches: combina los enfoques *Top Down*, *Bottom Up* y *Knowledge-based* en un único metamodelo (Von Mayrhauser y Vans, 1995). Propone que el entendimiento es construido concurrentemente en varios niveles de abstracción, a través del libre intercambio entre estas tres estrategias de comprensión.

Es importante tener en cuenta estas características debido a que proporcionan guías útiles para la elaboración de estrategias de Comprensión de Programas que asistan al programador en el durante el proceso de comprensión. Las teorías de MCs describen los conceptos esenciales empleados en este proceso y caracterizan el camino seguido por el programador para elaborar su conocimiento.

Todos los aspectos relativos a los MC mencionados en esta sección, evidencian la complejidad que implica desarrollar herramientas de comprensión que incluyan esta característica. Sin embargo, las herramientas que la tienen en cuenta, permiten: i) implementar las diferentes estrategias de aprendizaje, ii) seleccionar la estrategia más apropiada de acuerdo a los conocimientos subyacentes del programador, y iii) encontrar sentido y significado a las actividades del sistema.

2.2.2. Extracción de la Información

Una de las actividades iniciales de cualquier enfoque que se enmarque dentro del contexto de Comprensión de Programas, es la extracción de la información del sistema bajo estudio. Por esta razón, se utilizan técnicas que permiten obtener distintos tipos de datos relacionados con determinados aspectos de la estructura y/o el comportamiento del programa. Dichas técnicas se conocen en el contexto de CP como Técnicas de Extracción de Información (TEI).

Se puede decir que las TEI se subdividen en dos categorías dependiendo del tipo de información que se desee extraer: *TEI estática* y *TEI dinámica*. A continuación se

reseñan brevemente ambos enfoques:

TEI estática: permiten recuperar todos los atributos de los objetos definidos en el programa, ya sean tipos de datos, variables, funciones, entre otros; que se encuentran definidas en el código fuente del mismo. A menudo se utilizan técnicas de análisis sintáctico para obtener información estática del programa (De Lucia, 2001; Eisenbarth y cols., 2001; Rohatgi y cols., 2008).

TEI dinámica: posibilitan conocer los componentes del programa utilizados para una ejecución específica del sistema (Ball, 1999; Eisenbarth y cols., 2001; Cornelissen y cols., 2007). Las TEI dinámicas acotan el espacio de búsqueda, ya que en cada ejecución no se ven implicadas todas las secciones del código. Sin embargo, presentan dificultades debido al gran volumen de información que genera el programa en ejecución. Dichas técnicas son fundamentales para reunir información que capture el comportamiento de un sistema de software durante su ejecución.

Es importante mencionar que ambos tipos de técnicas son muy útiles para la elaboración de estrategias de comprensión. Esto se debe a que a través del análisis estático es posible construir representaciones del código del sistema que serían muy difíciles de recuperar con las técnicas de análisis dinámico, como por ejemplo el *Grafo de Dependencias del Sistema* (Sinha y cols., 1999). No obstante, las TEI dinámicas aportan información que no proveen las estrategias de análisis estático. Como ejemplos de esta última afirmación se pueden mencionar: *atribuir*[2] el *Grafo de Llamada a Funciones* con el número de veces que se ejecutó cada función para un escenario particular; completar el *Grafo de Llamada a Funciones* con las invocaciones que se realizaron a través de punteros, en lenguajes que permiten esta característica; entre otras (Eisenbarth y cols., 2001).

Es difícil determinar cual es la técnica más eficaz, ya que se podrían encontrar numerosos trabajos que presentan resultados interesantes usando alguna de las técnicas. Incluso diversos enfoques combinan ambos tipos de técnicas, logrando así destacar distintos aspectos del sistema mediante la información obtenida con técnicas dinámicas en conjunto con la información estática (Eisenbarth y cols., 2001; Gupta y cols., 1997; Berón, 2010).

[2]El uso del término atribuir se explicará más detalladamente en los próximos capítulos.

2.2.3. Administración de la Información

Una vez extraída la información debe ser administrada de manera apropiada. Dentro de las consideraciones que deben tenerse en cuenta en el contexto de Administración de la Información, se pueden identificar 2 aspectos principales: *la representación y manipulación de la información extraída* y *el filtrado de información irrelevante.*

El primer aspecto está relacionado con los tipos de estructuras soporte que más se adecuan para la inspección y representación de la información extraída. Dependiendo de los tipos de datos, la cantidad y la manera en que serán administrados, se deben seleccionar estructuras eficientes y robustas que brinden soporte a los datos y las operaciones que se realicen sobre las mismas (A. Aho y cols., 1988; Ali y Lhoták, 2012; Eisenbarth y cols., 2001; Grove y Chambers, 2001).

El segundo aspecto está relacionado con el volumen de información que se extrae de un sistema. El mismo puede llegar a ser de un tamaño considerable en muchos contextos. Esto se debe a que la dimensión de los sistemas ha crecido notablemente con el paso de los años. En la actualidad, la mayoría de los sistemas contienen un gran número de funciones, tipos de datos y estructuras que se relacionan entre sí. A esto también se le debe sumar la cantidad de información que produce un programa en ejecución. Esta información tiene que ser analizada con cuidado ya que el tamaño de la misma puede alcanzar una proporción inmanejable. Por lo tanto, en la mayoría de los casos es necesario tener en cuenta técnicas y estrategias de reducción de la información (Hamou-Lhadj y cols., 2005; Spärck Jones, 2007). Es importante que las mismas disminuyan significativamente la pérdida de artefactos de software sensibles a la lógica subyacente del sistema.

Se puede concluir que dependiendo de distintos factores como el volumen de información a manejar, la naturaleza de los datos a manipular, las operaciones necesarias para manejar las estructuras, entre otros, es conveniente seleccionar una estructura eficiente y robusta que de soporte a los datos y a las operaciones que se realicen sobre los mismos. Además también es necesario tomar en consideración el volumen de información que se maneja luego de la extracción. Teniendo en cuenta estos factores, se debe optar por una política de administración de la información adecuada y eficiente para cada caso en particular.

Una vez que se procesa y almacena la información, es necesario representar la misma de forma conveniente, con el objetivo de mostrar al programador la información procesada de la mejor manera posible. Para esto se debe tener en cuenta la Visualización de Software.

2.2.4. Visualización de Software

Una de los aspectos fundamentales a tener en cuenta en cualquier estrategia efectiva de comprensión es la manera en que se mostrará la información que ha sido procesada. La Visualización de Software (VS) es una disciplina de la Ingeniería de Software cuyo propósito es visualizar la información relacionada con los sistemas de software con el objetivo de simplificar el análisis y la comprensión de los mismos (Storey, Fracchia, y Müller, 1997; Stasko y cols., 1998). Actualmente, este aspecto se torna trascendental debido a que los sistemas de software son cada vez más grandes y complejos tornando su desarrollo y mantenimiento más engorroso (Mens y cols., 2002; Bassil y Keller, 2001; Ball y Eick, 1996).

Esta tarea generalmente se lleva a cabo a través de representaciones visuales denominadas en el contexto de VS como "vistas". Una vista es una representación de la información de un sistema que facilita la comprensión de un aspecto del mismo, es decir es una perspectiva del sistema. Las mismas son importantes debido a que actúan como puente cognitivo entre los conocimientos que posee el programador y los conceptos usados por el sistema (Berón, 2010; Mens y cols., 2002).

Claramente, la VS es uno de los aspectos más importantes de la CP. Es decir, pueden utilizarse buenas técnicas de extracción y administración de información; pero si no se tienen en cuenta técnicas de VS eficientes para cada caso, entonces la comprensión del sistema se verá entorpecida.

2.3. Modelos de Comprensión de Programas

A través de los años se han presentado numerosas propuestas con estrategias y técnicas que asisten al ingeniero de software en la difícil tarea de comprender un sistema. En esta sección se analizan los modelos de comprensión más utilizados por las estrategias y

técnicas planteadas. Para esto se toma como base el modelo de comprensión derivado del trabajo del grupo PCVIA (PCVIA, 2016). El uso de dicho modelo se basa en que este posee todos los elementos que deben ser considerados a la hora de analizar un modelo de comprensión.

El modelo del grupo PCVIA toma como base un extenso estudio y experiencia en el desarrollo de productos de comprensión (Berón, 2010; Lieberman y Fry, 1995; Von Mayrhauser y Vans, 1995), a través de los cuales se pudo comprobar que el principal desafío en la CP consiste en relacionar el Domino del Problema con el Dominio del Programa, como se muestra en la Figura 2.1. El primer Dominio, el del Problema, hace referencia a la salida del sistema. El segundo a las componentes de software usadas para producir dicha salida. Dicho modelo declara que entre el Dominio del Problema y el Dominio del Programa existe una relación real que será *re-construida* a nivel virtual con la finalidad de facilitar la comprensión.

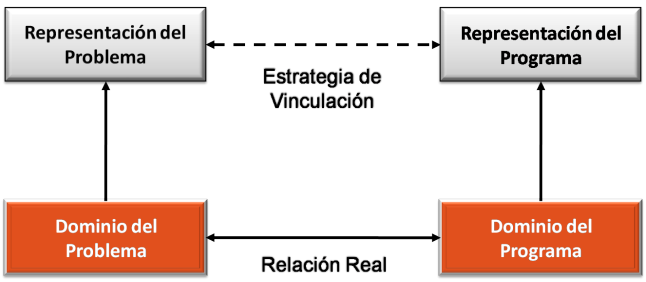

Figura 2.1: Modelo de Comprensión de Programas.

La construcción de este tipo de relación es llevada a cabo mediante los siguientes pasos:

- Construir una representación para el Dominio del Problema.
- Construir una representación del Dominio del Programa.
- Elaborar un procedimiento de vinculación.

Si bien no todas las propuestas de Comprensión de Programas siguen este modelo, el mismo es tomado como referencia ya que tiene en cuenta los componentes más utilizados en casi la totalidad de las propuestas. En los próximos apartados se analizan las variantes encontradas en la bibliografía referente.

2.3.1. Construir Representaciones para el Dominio del Programa

Intuitivamente se puede aseverar que para construir representaciones, primero es necesario extraer información del Dominio en cuestión. Desde el punto de vista del Dominio del Programa, todo este proceso se lleva a cabo a través del *análisis de código* (ver Figura 2.2). Binkley (Binkley, 2007) define al *análisis de código* como

> *proceso de extracción de información relativa a un programa a partir de su código fuente o los artefactos generados por el mismo (por ejemplo, desde un Java bytecode, trazas de ejecución, makefiles, etc.), utilizando herramientas automáticas.*

En la Figura 2.1 se exhibe el Modelo de CP explicado previamente, resaltando los componentes de dicho modelo que se ven afectadas por las tareas descriptas en el presente apartado.

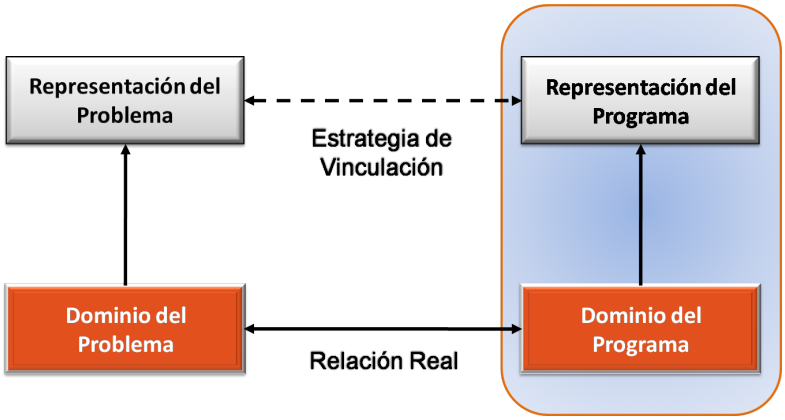

Figura 2.2: Construcción de representaciones para el Dominio del Programa. Representación basada en el Modelo de CP expuesto en la Figura 2.1.

Existen muchos métodos y herramientas desarrolladas para extracción de información del Dominio del Programa. Estas pueden ser clasificadas en base al tipo de información que extraen. De esta manera, se tienen TEI estáticas o dinámicas. La información estática es aquella que se obtiene a través del uso de técnicas de compilación tradicionales como por ejemplo: análisis lexicográfico, análisis sintáctico y análisis semántico del código fuente del programa. Este tipo de información es la que comúnmente manejan los

compiladores para determinar la "correctitud" de los programas, generar código, optimizar código, entre otras tantas. Este tipo de técnicas han sido comúnmente utilizadas en el ámbito de los lenguajes de programación. En este sentido, es posible mencionar trabajos ampliamente conocidos como el de Backhouse (Backhouse, 1979) o Aho y su grupo de investigación (A. V. Aho y cols., 2007), en donde se explican en detalle las distintas técnicas referenciadas en el párrafo precedente. También es importante destacar otro tipo de técnica estática muy utilizada en el ámbito de la CP, la misma se denomina *slicing* estático y fue propuesta por Weiser (Weiser, 1979).

Por otro lado, la información dinámica es la que se obtiene a través del uso de técnicas de análisis dinámico tales como instrumentación de código (Berón y cols., 2007b; Cuesta y García, 2008), *slicing* dinámico (Korel y Laski, 1988; X. Zhang y cols., 2003), *profiling* (Ball, 1999; Graham y cols., 1982), entre otras. Esta información es útil para analizar el funcionamiento del programa y establecer las partes del mismo utilizadas en resolver determinados aspectos del problema.

Claramente, los dos tipos de información brindan distintos enfoques y, por lo tanto, ambos son importantes. Es por esto que se puede combinar el análisis de ambos con el objetivo de mejorar la estrategia de comprensión. Existen diversos trabajos que sacan provecho a los dos tipos de información, combinando distintas técnicas.

En general, casi la totalidad de las propuestas que se enmarcan dentro de este modelo de comprensión generan representaciones visuales para abstraer la información extraída.

2.3.2. Construir Representaciones para el Dominio del Problema

Por otro lado, como se ha podido notar en la explicación del Modelo de Comprensión de la Figura 2.1, muchos trabajos emplean una representación del Dominio del Problema. La mayoría de las propuestas utilizan representaciones pre-establecidas como es el caso de las ontologías.

Sin embargo se han encontrado relativamente escasas propuestas de CP que realicen extracción de información del Dominio del Problema para posteriormente construir la representación del mismo (ver Figura 2.3). Uno de los principales factores que justifican

este hecho es que el Dominio del Problema no cuenta con la cantidad de artefactos que pueden encontrarse en el Dominio del Programa. Por lo general, el primero se encuentra plasmado en narrativas escritas en lenguaje natural u ontologías. También pueden considerarse la salida del sistema o su comportamiento como un artefacto del Dominio del Problema. Como puede observarse, los artefactos de este dominio no son fáciles de analizar y por consiguiente no es sencillo extraer información de los mismos. Es por este motivo que la mayoría de las propuestas que consideran el Dominio del Problema, lo hacen teniendo en cuenta representaciones ya definidas.

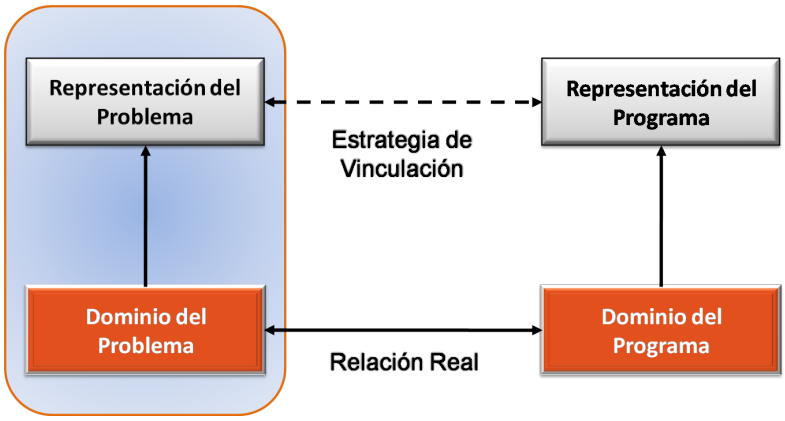

Figura 2.3: Construcción de representaciones para el Dominio del Problema en el Modelo de CP expuesto en la Figura 2.1.

2.3.3. Interconexión de Dominios

En la introducción a este capítulo se explicó en que consisten los MC, y el rol fundamental que ocupan dentro del contexto de la CP. Muchos autores han presentado distintas teorías cognitivas que establecen los elementos implicados en el proceso de entendimiento que utilizan los programadores mientras intentan comprender un programa (Brooks, 1983; Letovsky, 1986; Shneiderman y Mayer, 1979). La mayoría de las teorías propuestas afirman lo mismo: *Un programador comprende un programa, cuando este efectivamente puede relacionar el Dominio del Problema con el Dominio del Programa.* El modelo de comprensión expuesto en la Figura 2.1 al inicio de este capítulo tiene gran parte de su fundamento en dichos trabajos.

En la Figura 2.4 se ha resaltado los componentes del Modelo de CP utilizados en el

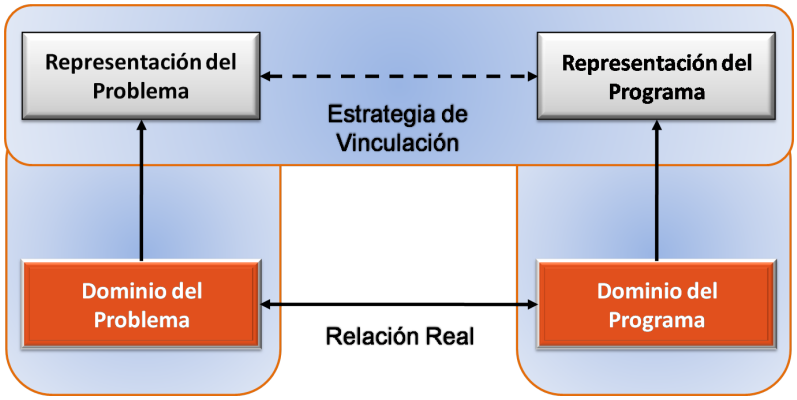

Figura 2.4: Construcción de representaciones para el Dominio del Problema y del Programa y proceso de vinculación en el Modelo de CP expuesto en la Figura 2.1.

proceso de interconexión.

2.4. Conclusión

La Comprensión de Programas es una disciplina de la Ingeniería de Software que ha tomado relevancia en las últimas décadas. La misma se centra en proponer técnicas, modelos, estrategias y herramientas que asistan al ingeniero de software en la difícil tarea de comprender un sistema.

Para diseñar e implementar estrategias de CP eficaces y robustas, es necesario tener en cuenta diferentes temáticas como lo son los diferentes Modelos Cognitivos utilizados por los programadores para comprender sistemas, las Técnicas de Extracción de Información empleadas para inspeccionar y obtener información relevante de los objetos de estudio, las estrategias de Administración de la Información que permiten definir las estructuras de base y el tipo de procesamiento que se le da a la información para lograr una gestión eficiente de la misma y la Visualización de Software que permite representar de distintas maneras la información extraída. Esto comprueba la complejidad que conlleva el diseño e implementación de estrategias de comprensión ya que requiere de un análisis interdisciplinario entre las distintas temáticas previamente mencionadas.

Otro aspecto fundamental es el modelo de comprensión que se propone para la estrategia, el cual subyace a todas las temáticas mencionadas en los párrafos precedentes. A través de los distintos trabajos es posible destacar ciertos modelos de comprensión a

los cuales se adhieren las diferentes propuestas.

En el próximo capítulo se brinda los trabajos relacionados al propuesto en este informe doctoral, teniendo como principal criterio la clasificación expuesta en este capítulo.

Capítulo 3

Estado del Arte de Herramientas de Comprensión de Programas

Si he podido ver más lejos, fue parándome en hombros de gigantes.

— Isaac Newton

Del capítulo anterior es posible concluir que existen distintos modelos de comprensión los cuales establecen, usando un nivel alto de abstracción, cómo se facilitará al ingeniero de software la comprensión del sistema bajo estudio. Además también se puede inferir que para elaborar estrategias de CP robustas y eficaces, es necesario reconstruir la relación entre el Dominio del Problema y el Dominio del Problema.

En las últimas décadas se han presentado diversos enfoques que se adhieren a distintos modelos de CP. Sin embargo, el foco de este trabajo está centrado en aquellas propuestas que han intentado de alguna manera crear una o varias vinculaciones entre los Dominios previamente mencionados.

Dado que las propuestas relacionadas a esta tesis doctoral abarcan diversas temáticas, los trabajos relacionados se exponen tomando como base la siguiente estructuración:

- Sección 3.1: propuestas que abordan la interconexión entre Dominios
- Sección 3.2: trabajos que introducen, usan y/o implementan técnicas similares a la

aproximación planteada en esta tesis doctoral.

3.1. Propuestas que Abordan la Interconexión entre Dominios

A continuación se mencionan los trabajos más relevantes que se enmarcan dentro del modelo de comprensión descrito en la Subsección 2.3.3. Este modelo de comprensión propone que para lograr un completo entendimiento del sistema es necesario vincular el Dominio del Problema con el Dominio del Programa. Los trabajos abajo referenciados están fuertemente relacionados con este modelo de comprensión de la misma forma que esta tesis doctoral.

Para una mejor exposición de los trabajos relacionados en este contexto, resulta conveniente clasificar los mismos de acuerdo a las técnicas utilizadas para analizar la información de los sistemas: análisis estático, análisis dinámico y enfoques híbridos. Dicho criterio ha sido ampliamente utilizado en el entorno de CP para destacar las distintas propuestas en la disciplina.

3.1.1. Propuestas que emplean Análisis Estático

Algunos trabajos han propuesto en cierta manera interconectar Dominios usando información estática extraída del sistema bajo estudio.

Uno de los primeros enfoques que intentaban vincular los dominios fue propuesto por Biggerstaff y su grupo de investigación (Biggerstaff, 1989; Biggerstaff y cols., 1993). La propuesta fue implementada mediante una herramienta denominada *DESIRE*, desarrollado por *Microelectronics and Computer Consortium*. El objetivo de esta herramienta es recuperar información del diseño del sistema bajo estudio. Para esto utiliza *conocimiento informal*, como nombres de variables y/o comentarios; por otro lado lleva a cabo análisis más formales, con el objetivo de construir una jerarquía de conceptos que en conjunto describen un programa. Una de las características más importantes de *DESIRE* es que los conceptos tratados son relativos al Dominio del Problema y no del Programa. La herramienta además posee una característica experimental que utiliza tecnología de redes

neuronales en el proceso de reconocimiento y vinculación con los artefactos del Dominio del Programa.

Murphy y su grupo de investigación (Murphy y cols., 1995) desarrollaron un enfoque que permite a los ingenieros de software i) especificar modelos de alto nivel de un sistema bajo estudio y ii) definir cómo ciertas partes del código fuente del mismo se mapean con los elementos de dicho modelo. La estrategia utiliza TEI estáticas para obtener información respecto a artefactos de software del código fuente, tales como identificadores de funciones, llamadas entre funciones, entre otros. De esta manera se construye, a partir de dichos artefactos, una de las estructuras más empleadas en el contexto de Ingeniería Reversa: el *Grafo de Llamadas a Funciones (GLF)* [1]. Posteriormente se computa un modelo de reflexión (*Reflexion Model*) que usa esta estructura generada y otros datos adicionales, para determinar dónde concuerda y diverge respecto del modelo especificado por el ingeniero de software. Esta técnica ha sido utilizada en diversos trabajos relacionados con análisis de términos y ubicación de conceptos en el código fuente (Siff y Reps, 1999; Rajlich y Wilde, 2002; Streekmann, 2011). La propuesta fue implementada mediante una herramienta denominada *RMTool* (Murphy y cols., 1995, 1998). En los trabajos expuestos, los autores exhiben un conjunto de casos de estudio con sistemas de distintas características (como diversos tamaño, distintos lenguajes de programación, distintos propósitos, etc.), mostrando la asistencia que la herramienta provee al ingeniero de software para entender dichos sistemas.

De manera similar, Christl y su grupo de investigación (Christl y cols., 2007) proponen un enfoque usando técnicas de clustering para dar soporte al proceso de mapeo que implica el método de reflexión. El rol del algoritmo de clustering es identificar aquellas entidades en el código fuente para las cuales el mapeo sea directo al punto tal que se pueda automatizar el proceso. Por otra parte, proporciona al ingeniero de software las facilidades para que pueda determinar aquellos mapeos menos directos. Los autores desarrollaron una herramienta denominada *HuGMe* que implementa las características mencionadas previamente. Según se exhibe en el trabajo, la herramienta fue probada con 4 sistemas (2 subsistemas de una herramienta denominada Bahaus (Raza y cols.,

[1] Un *GLF* se define como $GLF = (P, E)$ donde $P = \{x/x \text{ es una función del sistema}\}$ y $E = \{(x, y)/x \in P \wedge y \in P \wedge x \text{ llama a } y\}$. El GLF se describirá más detalladamente en el Capítulo 5.

2006), el juego Tetris y la herramienta de CP SHriMP (Wu y Storey, 2000)). En general, el trabajo muestra buenos resultados ya que, si bien cierta proporción de entidades de los sistemas fueron agrupadas con asistencia del ingeniero de software, la mayoría fueron agrupadas de manera automática por la herramienta.

Otra de las primeras propuestas que empleaba exclusivamente análisis estático fue la de Rugaber (Rugaber, 1995). En dicho trabajo el autor manifiesta que la mayoría de las herramientas de comprensión responden a las preguntas *qué* y *cómo* pero no al *porqué*. Este afirma que las preguntas relativas al *porqué* relacionan las construcciones del programa con el problema que deberían resolver. Dicho de otra manera, la mayoría de las herramientas sólo abordan el Dominio del Programa, proporcionando poca atención al Dominio del Problema y la interconexión entre estos. Basándose en esta aproximación, este autor y su grupo de investigación fueron proponiendo distintas estrategias y herramientas para interrelacionar los Dominios antes referenciados (Clayton y cols., 1997; Parnin y Rugaber, 2012; Rugaber, 2000). Un claro ejemplo de estas es *Dowser* (Clayton y cols., 1998). Los autores definen a *Dowser* como un *framework* compuesto de distintas herramientas que en términos generales permiten: i) analizar el Dominio del Problema en busca de conceptos; ii) analizar estructuras de datos definidas por el usuario y la interacción entre las mismas; iii) analizar la interacción entre los módulos del programa; iv) extraer componentes de la arquitectura del sistema bajo estudio; v) interconectar los conceptos del Dominio del Problema extraídos en el punto i) con los artefactos extraídos en los puntos ii), iii) y iv).

Haiduc y Marcus (Haiduc y Marcus, 2008) presentan un analizador sintáctico simple con las acciones semánticas necesarias para la extracción de información informal desde el código fuente de programas Java y C++. Posteriormente, dicha información se procesa con el objetivo de filtrar términos relativos al dominio de estudio. Como entrada, la herramienta recibe un conjunto de *términos de dominio*. Un *término de dominio* se refiere a los conceptos existentes los cuales se pueden describir usando palabras fuertemente relacionadas con el Dominio del Problema. En este sentido, la propuesta de los autores permite identificar aquellos artefactos del código fuente que implementan, o están estrechamente vinculados, con los términos de interés del usuario. Es importante remarcar que para definir previamente un conjunto de *términos del dominio*, se debe tener un

amplio conocimiento del Dominio del Problema, ya que dichos *términos* forman parte del vocabulario del mismo.

3.1.2. Propuestas que emplean Análisis Dinámico

En este apartado se mencionan algunos trabajos que han propuesto estrategias para interconectar componentes de ambos dominios usando exclusivamente información dinámica.

Se podría identificar como una de las primeras aproximaciones de interconexión de dominios usando información dinámica a la propuesta de Wilde y Scully (Wilde y Scully, 1995). Los autores diseñaron el método *Software Reconnaissance method* mediante el cual han sido pioneros en el campo de localización de características (*feature location*). El método utiliza información dinámica para localizar ciertas características y establecer relaciones entre conceptos del Dominio del Problema y artefactos del código fuente del sistema bajo estudio. La temática ha despertado mucho interés en las últimas décadas dentro del contexto de CP. Wong y su grupo de investigación (Wong y cols., 2000) analizan *rodajas* (*slices*) de casos de prueba con el mismo objetivo. Eisenbarth y su grupo de investigación (Eisenbarth y cols., 2003) usan información dinámica obtenida de ciertos escenarios que invocan determinadas características de un sistema y a partir de la misma aplica análisis formal de conceptos.

Safyallah y Sartipi (Safyallah y Sartipi, 2006) proponen aplicar algoritmo de *pattern mining* a las trazas obtenidas para extraer los patrones secuenciales más frecuentes respecto de las características exploradas en cada traza. En general, el término *característica* a la cual hacen referencia los trabajos mencionados son conceptos específicos que describen alguna funcionalidad fácilmente observable durante la ejecución del sistema.

3.1.3. Propuestas que emplean Análisis Híbridos

Muchos de los enfoques que han propuesto interconexión de dominios han empleado enfoques híbridos, es decir, utilizan información estática y dinámica para vincular artefactos del Dominio del Problema con componentes del Dominio del Programa.

Es posible establecer en cierta forma, como una de las primeras aproximaciones a

esta temática específica, al trabajo de Lieberman y Fry (Lieberman y Fry, 1995). En dicho trabajo los autores destacan la relevancia que tiene relacionar los Dominios de Problema y Programa, con el objetivo de comprender un sistema en particular. Los mismos proponen un prototipo de herramienta denominada *Zstep*. Dicha herramienta posee un ambiente de depuración con ciertas características particulares que brindan soporte cognitivo al proceso de comprensión llevado a cabo por el programador. Algunas de estas características son: animación de la ejecución del programa, representaciones del programa, interacción entre los artefactos gráficos y el código que los genera, entre otras. A través de esta herramienta, es posible observar el comportamiento del programa durante su ejecución mientras se visualiza el código que genera dicho comportamiento. Además provee un panel en el que se muestran las variables que sean de interés para el usuario y la posibilidad de avanzar o retroceder la ejecución del programa bajo estudio. Si bien la estrategia planteada en este trabajo no puede ser considerada como una de las más robustas, es relevante destacar que fue una de las primeras tentativas en relacionar los Dominios y además remarcar la importancia de esta interconexión para consolidar la comprensión.

Por otro lado, Cross y su grupo de investigación (Cross II y cols., 1996) proponen *GRASP* (Graphical Representations of Algorithms, Structures and Processes), una herramienta de Ingeniería Reversa para la generación automática de Diagramas de Estructuras de Control (Cross II y cols., 1998) (o por sus siglas en inglés, CSDs) a partir de código fuente escrito en Ada 95. Dicha herramienta fue una de las primeras aproximaciones que luego se vieron plasmadas en *jGRASP*, la misma es un ambiente de programación que posee funcionalidades de CP, depuración y de análisis de programas escritos en C, C++, Ada, Java y VHDL. *jGRASP* propone mecanismos para interconectar los Dominios del Problema y Programa, a través de la combinación de información estática y dinámica, para determinados casos de estudio. Por ejemplo, cuando el Dominio del Problema consiste en la implementación de tipos de datos abstractos, jGRAPH muestra la salida del programa, los datos y partes del programa empleados para producirla. Para aplicaciones más grandes, actúa como un depurador. Dicha herramienta genera un conjunto de vistas importantes de forma automática (Cross II y cols., 1998; Hendrix y cols., 2004). Entre las principales se encuentran: *Diagramas de Estructuras de Control*

(CSDs); *Digramas de Clase de UML*[2]; *Visores de Vistas Dinámicas*; entre otras.

Por otra parte, es importante destacar el trabajo de ciertos investigadores del Grupo de Especificación y Procesamiento de Lenguajes (GEPL) de la Universidade do Minho (GEPL, 2017). Los mismos han dedicado tiempo y esfuerzo de investigación en diseñar e implementar estrategias de Comprensión de Programas. En este contexto, uno de los primeros trabajos fue presentado por Pereira y su grupo de investigación (Varanda Pereira y Henriques, 2001). En el mismo los autores presentan una herramienta para animación y visualización de programas denominado *Alma* (Alma-Team, 2007). Esta herramienta relaciona los Dominios del Problema y Programa de un sistema, independientemente del lenguaje de programación utilizado. *Alma* simula el comportamiento del sistema bajo estudio visualizando el código fuente que está siendo ejecutado, la salida producida por dicha ejecución y las estructuras de datos que están siendo utilizadas en el escenario de ejecución. La independencia de los lenguajes de programación se logra por medio de la especialización de un *front end* (Varanda Pereira y Henriques, 2001), que permite generar el mismo tipo de representaciones internas para distintos lenguajes. De esta manera *Alma* exhibe las mismas visualizaciones sin importar la naturaleza del lenguaje de programación. La relación alcanzada puede ser fuerte o débil dependiendo del método usado para especificar el Dominio del Problema y de la especialización del *front end*. Es relevante resaltar que dicha herramienta fue propuesta con propósitos educacionales. Posteriormente, Nuno Oliveira (Oliveira, 2009) extiende los conceptos aplicados en Alma para Lenguajes Específicos del Dominio (o por sus siglas en inglés, DSL) en una nueva aplicación denominada *Alma*[2] (Alma-Team, 2007).

Berón y su grupo de investigación (Berón y cols., 2007b; Berón, 2010; Berón y cols., 2010) proponen dos estrategias de interconexión que se ven plasmadas en la herramienta *Program Inspection and Comprehension System* (*PICS*)(Berón y cols., 2007a); las mismas son: i) *Simultaneous Visualization Strategy* (SVS) y ii) *Behavioral-Operational Relation Strategy*(BORS). La primera muestra los artefactos de software que producen el comportamiento del sistema a medida que el mismo se ejecuta. La segunda permite al usuario obtener *explicaciones* de los artefactos de su interés identificados durante la ejecución del sistema (A. Aho y cols., 1988) (con SVS). *PICS* está orientado a mejorar

[2]Para programas escritos en Java

la comprensión de sistemas escritos en lenguaje C. Lo interesante de esta herramienta es que realiza dos tipos de análisis, uno durante la ejecución del sistema (*en vida*, *SVS*) y otro luego de la ejecución (*post mortem*, *BORS*). Usan una estrategia similar, Fonseca (Fonseca y cols., 2008) propone una herramienta denominada *Web Application Viewer (WAV)* que muestra la relación entre los dominios permitiendo visualizar aquellos artefactos del código fuente que fueron utilizados para producir cierta salida usando *BORS*. Además presenta un esquema de visualización basado en iconos, diseñados con la finalidad de representar fielmente cada componente del programa, y las relaciones definidas entre ellos que indican como se ensamblan cada pieza de código. Mediante este esquema se proveen interesantes funciones de navegación que facilitan la inspección del sistema. Una característica relevante de *WAV* es la posibilidad de analizar diferentes lenguajes de programación tales como HTML, PHP, Java; etc.

Anma y su grupo de investigación (Anma y cols., 2002, 2004) proponen un enfoque que difiere de los expuestos anteriormente. Los autores presentan un sistema que genera *explicaciones* textuales (en términos del Dominio del Problema) del comportamiento del programa bajo estudio. Posteriormente dichas *explicaciones* son mapeadas con entidades visuales del Dominio del Problema de la aplicación. De esta manera, a medida que se van ejecutando las sentencias, es posible visualizar el comportamiento del programa por medio de *explicaciones* textuales y entidades visuales generadas a partir del Dominio del Problema. Para esto se debe definir un *modelo del Dominio del Mundo* [3] para cada Dominio que se quiera analizar (Anma y cols., 2004). En los trabajos citados no se especifica claramente en que consisten dichos *modelos*. Es importante destacar que el sistema fue implementado para programas escritos en PASCAL y además que el mismo fue concebido con propósitos educacionales.

Sato y su grupo de investigación (Sato y cols., 2008) presentan una herramienta denominada *ORCA*, que proporciona los mecanismos necesarios para entender la funcionalidad de las interfaces gráficas de usuario implementadas en lenguaje Java. *ORCA* logra relacionar los Dominios del Problema y Programa a través de: i) la extracción y representación de información estática y dinámica y ii) la utilización de un ambiente que visualiza diapositivas cuyo contenido son imágenes de la salida del sistema junto con el

[3] Los autores hacen referencia a un *modelo* para describir el Dominio del Problema.

correspondiente código fuente. De la misma manera que *PICS*, *ORCA* realiza dos tipos de análisis: *en vida* y *post mortem* (Sato y cols., 2008).

Más recientemente, Nuno Carvallo y su grupo de investigación (Nuno y cols., 2012; Carvalho y cols., 2015) proponen crear mapeos bidireccionales entre los conceptos del Dominio del Problema y artefactos del Dominio del Programa para asistir durante los proceso de comprensión. Para esto la estrategia plantea crear 3 ontologías que usará como base para el proceso de vinculación. La primer ontología es la del programa, que contiene información del código fuente y es extraída usando técnicas de análisis estático, procesamiento de lenguaje natural y técnicas de recuperación de información. La segunda ontología contiene, información referida a la ejecución del sistema (en caso en que sea posible). Por último utiliza una ontología del Dominio del Problema elaborada a partir de técnicas de análisis del dominio y recuperación de información tomando como base documentación y comentarios del código fuente. Posteriormente, el autor establece puentes semánticos entre las distintas ontologías utilizando un *mapeador de conceptos*. Una vez establecidos todos los mapeos entre las ontología, se usa una capa de razonadores sobre las ontologías para brindar información provechosa para los ingenieros de software.

3.2. Propuestas estrechamente relacionadas al trabajo doctoral

Durante las ultimas décadas se han propuesto numerosas estrategias para recuperar distintas características y vistas de los sistemas. Varias de estas propuestas han sido mencionadas en apartados anteriores donde se ha intentado clasificar las mismas de acuerdo a las TEIs y el tipo de información empleada. Sin embargo, también es importante considerar aquellos trabajos que han planteado aproximaciones utilizando técnicas y estrategias que están estrechamente relacionados con este trabajo. En los próximos apartados, mencionaremos aquellas propuestas dentro del contexto de Ingeniería Reversa que específicamente:

- han utilizado técnicas de clustering;
- han inferido modelos de casos de uso o

- han empleado métricas para asistir al proceso de comprensión.

La mayoría de las propuestas mencionadas en los apartados subsiguientes adhieren al modelo de comprensión descrito en el apartado 2.3.1. El mismo plantea generar abstracciones y vistas de software para asistir al ingeniero de software durante la CP sin tener bajo consideración el Dominio del Problema. Sin embargo, más allá del modelo de comprensión al cual se adhieran las propuestas, es interesante destacar las distintas técnicas utilizadas en cada trabajo.

3.2.1. Propuestas que emplean Técnicas de Clustering

En el contexto de CP, las técnicas de clustering de software permiten agrupar entidades de un sistema de software, tales como métodos, clases, archivos fuentes, en componentes que representan abstracciones de más alto nivel. El principal objetivo es asistir al proceso de comprensión de un sistema mediante una vista particular del sistema.

Con este fin, algunos trabajos usan criterios basados en propiedades estructurales de las distintas representaciones construidas a partir del código fuente para realizar los agrupamientos y de esta manera generar abstracciones del sistema. Como claro ejemplo de este tipo de propuestas se pueden mencionar al trabajo de Belady y Evangelisti (Belady y Evangelisti, 1981), los cuales proponen utilizar ligaduras de datos para construir un grafo entre procedimientos y así agrupar los mismos en módulos. Por su parte, Hutchens y Basili (Hutchens y Basili, 1985), extienden el enfoque anterior para producir *dendrogramas*[4] que representan jerarquías de módulos. Schwanke (Schwanke, 1991) fue uno de los primeros autores en proponer una técnica de clustering basada en que los grupos obtenidos tengan alta cohesión y bajo acoplamiento. Dicho autor desarrolló la herramienta *ARCH* que implementa esta técnica. De manera similar Muller y su grupo de investigación (Müller y cols., 1993) presentan un enfoque que permite agrupar módulos teniendo en cuenta distintas métricas estructurales en conjunto con los nombres de las entidades a ser agrupadas. Mancordis y su grupo de investigación (Mancoridis y cols., 1998) tratan el agrupamiento como un problema de optimización utilizando algoritmos

[4] Un *dendrograma* es un tipo de representación gráfica en forma de árbol que organiza los datos en subcategorías hasta llegar al nivel de detalle deseado. Este tipo de representación permite apreciar claramente las relaciones de agrupación entre los datos e incluso entre grupos de ellos.

genéticos y *hill-climbing* para lograr óptimos locales para una función. Dicha función utiliza como principal criterio que los agrupamientos tengan alta cohesión y bajo acoplamiento. Bahuer y Trifu (Bauer y Trifu, 2004) proponen un algoritmo que agrupa módulos combinando el uso de i) los subgrafos derivados del *minimum spanning tree* del grafo de dependencia del sistema bajo estudio y ii) detección de pistas de la arquitectura del sistema por medio de correspondencia de patrones. Xiao y Tzerpos (Xiao y Tzerpos, 2005) proponen hacer uso de la información dinámica a través de las trazas en vez de detectar interrelaciones estáticas. Los autores afirman que las técnicas de clustering deben nutrirse de información dinámica. Zhang y su grupo de investigación (Q. Zhang y cols., 2010) proponen un algoritmo híbrido que usa como base el *Grafo de Clases Pesado* (WDCG, por sus siglas en inglés) que contiene información estática y dinámica (la relaciones entre clases y las veces que se "relacionan"). El enfoque usa dos algoritmos de clustering: jerárquico y *minimum spanning tree* que toman como base el acoplamiento y el peso de los arcos. Erdemir y su grupo de investigación (Erdemir y cols., 2011) presentan un enfoque que usa un algoritmo de clustering para detección de "estructuras de comunidades", diseñado específicamente para grafos de redes sociales. Los autores exponen la viabilidad de la aplicabilidad de dicho algoritmo a los diagramas de clases de sistemas orientados a objetos mostrando que los mismos poseen similitudes en la estructura.

En este contexto, Shtern y Tzerpos (Shtern y Tzerpos, 2012) presentan un amplio estado del arte exponiendo un gran número de este tipo de técnicas. Es importante destacar que este las mismas sólo toman en consideración exclusivamente información estructural de las representaciones utilizadas para modelar los sistemas, por ejemplo, grafos de llamadas a funciones, estructura de archivos, uso compartido de datos, entre otros. La mayoría de estas técnicas han sido concebidas en otras áreas donde no existe mayor información que la provista por la estructura a ser analizada (por ejemplo, biología, matemática, redes sociales, etc). Este tipo de enfoques, donde la información sustancial se encuentra en las relaciones entre componentes de una representación del sistema (por ejemplo, relaciones entre nodos en un grafo), dejan de lado información relevante en un contexto de Ingeniería Reversa. Como claro ejemplo de lo antes mencionado se pueden considerar las características de la entidad que representa el nodo. Es decir, si el nodo representa una función, entonces sería poco conveniente dejar de lado ciertos aspectos

inherentes a la función, tales como información de los parámetros, tipo de retorno de la función, indicadores de complejidad de la función, relación con la interfaz gráfica del sistema, etc.

Por otra parte, también se han desarrollado técnicas de clustering de software que, a diferencia de las anteriores, consideran otro tipo de información del sistema además de la estructural. A continuación se exponen algunos trabajos enmarcados dentro de este contexto.

Anquetil y Lethbridge (Anquetil y Lethbridge, 1997) introducen una técnica de clustering para agrupar archivos usando como principal criterio las convenciones utilizadas para nombrar los mismos.

Bowman y Holt (Bowman y Holt, 1998) proponen una estrategia de clustering que usa como base además de los criterios estructurales comunes en otras investigaciones, información respecto a los equipos de desarrollo con el objetivo de extraer la arquitectura del sistema. Los autores afirman que la arquitectura esta fuertemente relacionada con la distribución de los grupos de desarrolladores de cada módulo del sistema.

Tzerpos y Holt (Tzerpos y Holt, 2000) presentan un algoritmo denominado *ACDC* que emplea distintos tipos de patrones o reglas de agrupamiento para determinar descomposiciones basadas en distintas propiedades relativas a CP. El algoritmo aplica sistemáticamente patrones para detectar subsistemas en la estructura del software. Por otra parte implementa un mecanismo de adopción de huérfanos para asignar nodos sueltos a los subsistemas apropiados ampliamente utilizado en otras propuestas.

Maletic y Marcus (Maletic y Marcus, 2001) usan una técnica de recuperación de la información denominada *Latent Semantic Indexing (LSI)* para agrupar archivos del sistema (aunque los autores afirman que se puede usar elementos de distinta granularidad como funciones, clases, etc.) usando información informal y estructural del sistema. La base del algoritmo se centra en que las partes de un sistema de software que usan términos similares están interrelacionados. Los autores proponen utilizar esta técnica para identificar tipos de datos abstractos.

Andritsos y Tzerpos (Andritsos y Tzerpos, 2005) presentan un algoritmo de agrupamiento jerárquico que utiliza conceptos de teoría de la información en conjunto con información sobre la estructura de directorio de los archivos y un mapeo entre los ele-

mentos del código fuente y los recursos humanos que desarrollaron el sistema.

Beyer y Noack (Beyer y Noack, 2005) proponen agrupar artefactos de software (los mismos no están limitados sólo a elementos del código fuente) teniendo en cuenta los cambios históricos realizados a los mismos. Es decir, agrupa artefactos de software que son frecuentemente alterados en conjunto.

Kuhn y su grupo de investigación (Kuhn y cols., 2005) proponen agrupar entidades del código fuente (ya sean clases, métodos, o subsistemas completos) utilizando información informal. De la misma manera que Maletic y Marcus, usan *LSI* para relacionar términos de información informal con las entidades que serán agrupadas.

Christl y su grupo de investigación (Christl y cols., 2007) proponen una técnica de clustering usando *reflexion methods*. Para esto se utilizan modelos que representan el Dominio del Problema cuyas entidades serán mapeadas con agrupamientos de artefactos de software extraídos desde el código fuente. En pocas palabras, este enfoque utiliza funciones de similaridad que comparan las semejanzas entre los artefactos de software y las entidades del modelo utilizado.

3.2.2. Obtención de Casos de Uso

También existen trabajos que proponen extraer información del sistema con el objetivo de obtener y/o analizar casos de uso de un sistema determinado. Una de las principales ventajas de utilizar casos de uso, en el contexto de Ingeniería Reversa, es que los mismos son fáciles de entender para todos los involucrados en los procesos de desarrollo o MES. Por lo tanto, se presenta como un medio de comunicación conveniente en el entorno (Salah y cols., 2006; L. Zhang y cols., 2006; Budgen y cols., 2011).

Bojic y Velasevic (Bojic y Velasevic, 2000) integran técnicas de análisis estático y dinámico. Los autores analizan conjunto de casos de uso sustanciales, es decir, las funcionalidades más importantes del sistema de manera manual y extraen los correspondientes casos de prueba. Las trazas de ejecución obtenidas a partir de los casos de prueba son mapeados con entidades en el código fuente. Aplicando análisis de conceptos sobre el código fuente, las entidades detectadas en las trazas que implementan funcionalidades similares son agrupadas y vinculados a los casos de uso correspondientes. Salah y su gru-

po de investigación (Salah y cols., 2006) presentan un enfoque semejante, sin embargo no utilizan modelos UML sino que crean sus propias vistas con el objetivo de proveer diversos niveles de abstracción. Al utilizar diversos niveles de abstracción, el enfoque permite una visión arquitectural, pero que puede ser detallada a nivel de código fuente de acuerdo a la necesidad del ingeniero de software.

El-Ramly y Sorenson (El-Ramly y cols., 2002) plantean obtener casos de uso para representar el comportamiento del sistema usando trazas generadas a partir de la interacción del usuario con el sistema. Utilizan data mining y pattern matching para explorar estas trazas con el objetivo de analizar las tareas más importantes para el usuario. Cuando se encuentran patrones relevantes, son enriquecidos por el ingeniero de software con información semántica y tomando los mismos como base, se construyen los casos de uso.

Zhang y su grupo de investigación (L. Zhang y cols., 2006) presentan un enfoque para asociar código fuente a los casos de uso a través de análisis estático. El método es semi-automático y genera un grafo de llamadas que considera el flujo de control de los programas. Luego se usan heurísticas para filtrar funciones de bajo nivel. Por último, arman el modelo de casos de uso. El principal criterio de detección de casos de uso son las bifurcaciones en el código y para la detección de dependencias entre estos se analizan las llamadas a funciones. Si y su grupo de investigación (Si y cols., 2013) presentan una aproximación parecida, sólo que construyen un grafo de llamadas a funciones con bifurcaciones. Luego realizan un pesaje sobre el mismo de acuerdo a una métrica de complejidad. Posteriormente extraen trazas de manera estática (simulan ejecución) para capturar todas los posibles caminos de ejecución. Durante este proceso, se asiste al ingeniero de software para que: i) determine los distintos escenarios; ii) determine que escenarios corresponden a un mismo caso de uso; iii) identifique los actores, la dependencia entre los casos de uso y los tipos de dependencia (relaciones include, extend o generalización). Li y su grupo de investigación (Li y cols., 2007) proponen un enfoque similar al de los dos anteriores sólo que usan información dinámica.

Dugerdil y Repond (Dugerdil y Repond, 2010) usan información dinámica y técnicas de clustering para generar abstracciones de diagramas de secuencias que representan especificaciones de casos de uso particulares con el objetivo de ayudar al ingeniero de software a comprender los sistemas. El algoritmo de clustering construye un vector por

cada clase que refleja los sectores de la traza donde dicha clase ha sido utilizada. Luego usa una función de similaridad que toma como base dicho vector para comparar clases y determinar si las mismas deben ser agrupadas. Con el mismo objetivo Ziadi y su grupo de investigación (Ziadi y cols., 2011) proponen una estrategia que comprende los siguientes pasos: 1) capturar varias trazas de ejecución de programas java de acuerdo a casos de uso relevantes, 2) realizar unión de trazas usando un algoritmo denominado *k-tail* y la representación Labeled Transition System (LTS) y 3) Extracción de los diagramas de secuencia para especificación de casos de uso.

Zhou y su grupo de investigación (Zhou y cols., 2009) presentan un enfoque para detectar flujos principales y alternativos en cada caso de uso. En primer lugar extraen el grafo de flujo de control del programa. Luego la estrategia analiza el código buscando todas las interacciones con el usuario y este debe determinar cuales son críticas. Además la estrategia permite que el usuario defina prioridades para ciertas tareas las cuales son posteriormente analizadas para asistir en la determinación de los próximos casos de uso.

Pereira y su grupo de investigación (Pereira y cols., 2011) proponen extraer el modelo de casos de uso extrayendo información estática y dinámica del sistema. En primer lugar, transforman el código fuente en una representación que sólo refleja las llamadas a métodos. Posteriormente, ejecutan un algoritmo que aproxima el conjunto de casos de uso de manera estática (consideran a cada método un caso de uso). Finalmente, ejecutan un conjunto de reglas que permiten detectar: i) las dependencias entre los casos de uso; ii) el tipo de relación (generalización, include o extend); iii) casos de uso de más alto nivel y iv) flujos principales y alternativos dentro de cada caso de uso (usa información dinámica). Las reglas se aplican usando transformaciones de modelos, para esto los autores definen un framework de Ingeniería Reversa basado en MDA.

Dugerdil y su grupo de investigación (Dugerdil y Sennhauser, 2013) platean extraer los casos de uso de un sistema usando información dinámica. Los autores definen un nuevo formato para representar trazas de ejecución, el mismo permite especificar un árbol de decisión dinámico (representa las alternativas de ejecución dentro de la traza) para cada caso de uso. Además se propone un algoritmo de reducción de dicho árbol en conjunto con la identificación de flujos alterados por el usuario. De esta manera el

algoritmo posibilita la identificación de flujos principales y alternativos en la descripción de cada caso de uso.

3.2.3. Definición de Métricas Específicas para Comprensión de Programas

Esta subsección menciona algunos trabajos y sus correspondientes posiciones considerando los enfoques más relevantes que han utilizado/definido métricas en el contexto de Ingeniería Reversa.

En las últimas décadas, muchos trabajos han empleado métricas con diferentes propósitos, a modo de ejemplo es posible referenciar los siguientes:

- caracterizar metodologías de desarrollo (Shawky y Abd-El-Hafiz, 2016);
- refactorizar clones en el código (Higo y cols., 2008);
- predecir clases propensas a modificación (Elish y Al-Rahman Al-Khiaty, 2013);

Más específicamente, también se han definido e incluso implementado herramientas que estiman métricas en el contexto de Comprensión de Programas. En la mayoría de los casos se han propuesto variantes de las métricas de mantenibilidad (Coleman y cols., 1994; Chidamber y Kemerer, 1994; Wani y Gandhi, 1999) apuntando a un índice de *comprensibilidad*. Desde un punto de vista pragmático, es posible concebir a ambas métricas como la misma, ya que en general, un programa difícil de mantener muy posiblemente sea difícil de entender. Por ejemplo, un alto grado profundidad en el árbol de herencia de un sistema dificultará el mantenimiento y también la comprensión del mismo. Esto se debe a que a la hora de entender el funcionamiento y la estructura del sistema es necesario tener en cuenta factores como atributos y métodos heredados, polimorfismo, etc. que no suelen ser fáciles de detectar observando el código. Dentro de este contexto, es posible enmarcar los siguientes trabajos:

- Varios autores proponen usar métricas previamente definidas en el contexto de Ingeniería de Software, para inferir un grado de *comprensibilidad* para los sistemas, por ejemplo *índice de mantenibilidad, métodos pesados por clase, números de hijos,*

falta de cohesión en los métodos, complejidad ciclomática, etc. Como claros ejemplos de este tipo de enfoques es posible mencionar a Mathias y su grupo de investigación (Mathias y cols., 1999), Kasto y Whalley(Kasto y Whalley, 2013), Austin y Samadzadeh(Austin y Samadzadeh, 2005), la herramientas *Imagix4D* (Imagix-Corporation, 2016), *CodeCrawler* (Demeyer y cols., 1999), *Bauhaus* (Raza y cols., 2006), entre otros.

- Zuse (Zuse, 1993) propone un conjunto de criterios y operadores para describir el grado de comprensión de un sistema en términos de axiomas.

- Algunos trabajos proponen métricas específicas de complejidad de software con el objetivo de medir la comprensibilidad y/o predecir el grado de mantenibilidad de un producto de software (Poulin, 1994; Ahamad y Verma, 2016; Anderson y Zarins, 2005).

- Rilling y Klemola (Rilling y Klemola, 2003) definen ciertas métricas que en conjunto con técnicas de slicing permiten medir la comprensibilidad de un programa en término de sus identificadores.

- Power y Malloy (Power y Malloy, 2004) llevan el dominio de las métricas al nivel de las gramáticas de los lenguajes de programación para determinar la complejidad de los mismos a la hora de definir herramientas de CP.

- Yin y su grupo de investigación (Yin y cols., 2010) proponen dirigir el proceso de comprensión mediante una métrica que permite identificar aquellos métodos difíciles de comprender y las dependencias entre los mismos.

No obstante, hay muy pocas aproximaciones que proponen métricas con el principal objetivo de inferir la importancia relativa de los componentes del código fuente con respecto a la lógica subyacente del sistema. Los trabajos más referenciados mencionados previamente han propuesto métricas que no se pueden utilizar en este contexto ya que los mismos fueron diseñados con propósitos diferentes. Los enfoques más relacionados a este concepto proponen identificar "clases importantes" en sistemas desarrollados usando lenguajes orientados a objetos.

En el caso de Zaidman y su grupo de investigación (Zaidman y cols., 2005; Zaidman y Demeyer, 2008) hacen uso del algoritmo HITS (Kleinberg, 1999) que emplea una métrica de acoplamiento para detectar *distribuidores (hubs)* y *autoridades (authorities)* dentro de un grafo. Para construir el grafo combinan técnicas de *web mining* con análisis dinámico y estático. Los autores afirman que aquellas clases que cumplen roles de *pivotes* en el grafo son las más relevantes para la lógica subyacente del sistema. Por su parte, Kamran y su grupo de investigación (Kamran y cols., 2013, 2016) proponen un enfoque similar diseñando una métrica de acoplamiento que tiene algunas variantes para mejorar la detección de clases importantes. Şora (Şora, 2015) y Meyer (Meyer y cols., 2014) proponen adaptaciones a los algoritmos *PageRank* y *k-core* respectivamente, para elaborar rankings de las clases más importantes. Con el mismo propósito Thung y su grupo de investigación (Thung y cols., 2014) utilizan técnicas de *machine learning* combinadas con métricas de diseño y networking durante el proceso de aprendizaje. Steidl y su grupo de investigación (Steidl y cols., 2012) realizan un estudio para encontrar la mejor combinación de métricas de centralidad en un grafo estático para detectar las clases esenciales de un sistema.

Siguiendo estrategias similares a las mencionadas anteriormente, Perin y su grupo de investigación (Perin y cols., 2010) extienden la aplicación del algoritmo de *PageRank* a artefactos más "atómicos" del código fuente, como por ejemplo métodos y/o atributos. De manera similar, Robillard (Robillard, 2005) introduce un enfoque que analiza la topología del grafo de dependencias de un sistema para detectar componentes relevantes del mismo. Para esto requiere un conjunto de elementos de interés para el ingeniero de software. Mediante esta información calcula métricas de acoplamiento para determinar mediante el uso de grados de certeza los elementos más relevantes para el programador.

3.3. Discusión y Conclusión del Capítulo

Como es posible analizar, los enfoques relacionados al propuesto en este trabajo doctoral son numerosos y desarrollan diversas aristas dentro del contexto de Comprensión de Programas.

En la Sección 3.1 se expusieron distintas propuestas que asisten al ingeniero de soft-

ware en la comprensión de programas usando distintos tipos de enfoques. La mayoría de estos proponen vincular términos del Dominio del Problema con artefactos del código fuente del sistema (Wilde y Scully, 1995; Murphy y cols., 1995; Haiduc y Marcus, 2008; Christl y cols., 2007). Es relevante destacar que este tipo de vinculación es relativamente débil ya que implica que el ingeniero de software deba buscar aquellos términos específicos del Dominio del Problema para comenzar la vinculación. Esto conlleva poseer un profundo conocimiento del Dominio del Problema de la aplicación y, bajo ciertas circunstancias, entender algunos elementos en el código fuente.

Otros enfoques son dependientes de ciertos artefactos esenciales para que la estrategia sea efectiva. Por ejemplo en el caso de la herramienta derivada del trabajo de Rugaber (Rugaber, 1995; Clayton y cols., 1997; Parnin y Rugaber, 2012) posee dos desventajas significativas en su estrategia: i) no proveen vistas provechosas de la información extraída desde los distintos Dominios y ii) la documentación provista por el sistema bajo estudio debe ser completa y consistente ya que el análisis del Dominio del Problema se basa mayormente en la inspección de la misma.

Otros enfoques necesitan un modelo del Dominio del Problema el cual es difícil de construir y es poco usual encontrarlo en el contexto del sistema que se está analizando. Además, en la mayoría de este tipo de propuestas, la eficacia general de la estrategia estará supeditada a la precisión con la que el ingeniero de software construya dicho modelo. Como claros ejemplos de estos trabajos se pueden mencionar las siguientes propuestas:

- Anma y su grupo de investigación (Anma y cols., 2002) proponen una estrategia que se basa en el diseño de modelos del Dominio a donde puede ser aplicado dicho lenguaje. Los autores han manifestado que es imposible abarcar todos los posibles modelos del Dominio a donde se puede aplicar el lenguaje PASCAL. Por lo tanto es necesario definir el estado del universo por cada vez que se va a utilizar la herramienta para un Dominio en particular.

- Carvalho y su grupo de investigación (Carvalho y cols., 2015) en su estrategia requieren de una ontología del Dominio del Problema.

- La propuesta de Christl y su grupo de investigación (Christl y cols., 2007) recibe

como entrada un modelo de la arquitectura del sistema establecida por el ingeniero de software.

A los aspectos mencionados previamente en la sección, se le debe sumar que escasas aproximaciones que plantean vincular ambos dominios (descriptas en la sección 3.1) proponen/utilizan los siguientes enfoques:

Generar modelos de casos de uso. En el contexto de Ingeniería de Software este tipo de modelo ha sido ampliamente utilizado con diferentes propósitos en distintos trabajos de investigación (Budgen y cols., 2011; Salah y cols., 2006). Una de las ventajas más valiosas de este tipo de modelos, es que son fáciles de entender por todos los involucrados en el desarrollo de un sistema de software y los correspondientes procesos de reingeniería (Bojic y Velasevic, 2000).

Implementar técnicas de clustering enfocadas a CP. A lo largo de los años las técnicas de clustering han demostrado ser efectivas en el contexto de CP para generar abstracciones de los sistemas de software (Shtern y Tzerpos, 2012). La única herramienta de CP mencionada en la Sección 3.1 que utiliza técnicas de clustering es la de Christl y su grupo de investigación (Christl y cols., 2007).

Calcular métricas de software orientadas a CP. Dentro el contexto de CP, las métricas han sido utilizadas para obtener información del sistema bajo estudio. Por medio de la cuantificación de las características relevadas, es posible destacar de manera más precisa los distintos aspectos del sistema para inspeccionar su naturaleza (Mathias y cols., 1999; Zaidman y Demeyer, 2008).

En la Sección 3.2 se presentan aquellos trabajos que están estrechamente relacionados con esta tesis doctoral desde tres temáticas distintas: algoritmos de clustering, obtención de casos de uso y métricas de software. Teniendo en cuenta lo mencionado en dicha sección, es posible destacar ciertos aspectos que diferencian estas propuestas de la estrategia planteada en esta investigación doctoral. En los próximos apartados se analizan las debilidades encontradas en los trabajos relacionados, de acuerdo a cada temática planteada.

Algoritmos de clustering

Se mencionaron varios trabajos que han propuesto obtener abstracciones como arquitecturas del sistema (Ducasse y Pollet, 2009; Garcia y cols., 2013) o relaciones entre módulos usando técnicas de clustering que empleaban información estructural tal como llamadas a funciones, uso de variables, uso de clases, etc. Sin embargo, dichas propuestas descartan información inherente al software y los lenguajes de programación. De hecho, muchas de estas han sido introducidas en el contexto de otras disciplinas donde la información disponible se limita a aspectos estructurales de una representación. En el contexto de Ingeniería de Software, cierta información inherente a las entidades representadas en las estructuras son relevantes; como por ejemplo los parámetros de cada función, o en caso en que el sistema sea Orientado a Objetos, determinar si el método es de clase o de instancia, si es público o privado. Las técnicas de clustering de software no deberían descartar este tipo de información valiosa a la hora de agrupar artefactos con propósitos de CP.

Por otra parte, también se describió un conjunto de aproximaciones que utilizaban otro tipo de información inherente a los sistemas de software. Si bien varios de los trabajos mencionados en este contexto proponen alternativas distintas a aquellas que sólo emplean información estructural, la efectividad de sus resultados depende fuertemente de la existencia y claridad de cierto tipo de información. Como por ejemplo equipos de desarrollo (Bowman y Holt, 1998; Andritsos y Tzerpos, 2005), cambios históricos durante el desarrollo y/o mantenimiento del sistema (Beyer y Noack, 2005), identificadores de artefactos de software en el código fuente e información informal (Maletic y Marcus, 2001; Kuhn y cols., 2005), modelos del Dominio del Problema (Christl y cols., 2007), entre otros. Tomar como base exclusivamente este tipo de información, puede resultar en la obtención de descomposiciones erróneas del sistema ya que frecuentemente no se encuentra actualizada. Además es importante destacar que ésta información suele no estar presente en el proyecto o no estar del todo clara. Por otra parte, en el contexto de análisis de información informal, se debe sumar las dificultades que implica el tratamiento de lenguaje natural (consideración de sinónimos, verbos conjugados, errores ortográficos, etc.).

Uno de los enfoques más interesantes es propuesto por Tzerpos y Holt mediante el algoritmo *ACDC* (Tzerpos y Holt, 2000). Los autores afirman que, dentro del contexto de Comprensión de Programas, es más importante lograr que un agrupamiento ayude a entender el sistema antes que "maximizar una métrica". Proponen ciertos factores a los cuales se les debe dar importancia, como mapeo de patrones y el nombramiento efectivo de los clusters (Shtern y Tzerpos, 2012). El algoritmo es reconocido en el contexto de la disciplina demostrando ser uno de los más efectivos (Garcia y cols., 2013; Lutellier y cols., 2015). No obstante, se ha cuestionado la trivialidad de varios de los patrones presentados por el mismo en conjunto con los resultados obtenidos para ciertos casos de estudio (Wu y cols., 2005). Por otra parte, si bien el enfoque resulta interesante en diversos aspectos, los autores no incorporan en la propuesta información relativa al Dominio del Problema.

Obtención de casos de uso

También se mencionaron ciertos trabajos que han intentado inferir casos de uso para asistir en los distintos procesos ingenieriles tales como, reestructuración, reingeniería, Ingeniería Reversa, etc.

Todas las propuestas difieren en distintos aspectos de la desarrollada en esta tesis y presentan ciertas desventajas en común, por ejemplo, muchos de los trabajos no construyen las relaciones entre casos de uso, sino que vinculan artefactos del código a determinados casos de uso ya identificados (Bojic y Velasevic, 2000; Dugerdil y Sennhauser, 2013; Si y cols., 2013). Además, la mayoría de las propuestas requieren de un usuario con un amplio conocimiento del sistema para: i) detectar casos de uso esenciales (Bojic y Velasevic, 2000; Dugerdil y Sennhauser, 2013; Si y cols., 2013); ii) identificar propiedades en los modelos de casos de uso como tipos de relaciones (include, extend o generalizaciones), incluso actores o agrupamiento de casos de usos (Dugerdil y Sennhauser, 2013; Pereira y cols., 2011); o iii) insertar acciones semánticas que ayuden a detectar componentes en el modelo (El-Ramly y cols., 2002). Otras propuestas detectan casos de uso usando como principal criterio el flujo de control en el código (L. Zhang y cols., 2006; Si y cols., 2013). Sin embargo, no es directa la relación entre los diferentes flujos de control y los casos de uso.

Una de las principales desventajas que tienen en común todas las propuestas es

que no consideran ciertos artefactos importantes presentes en la mayoría de los sistemas actuales, como lo son los componentes de la GUI. Estos componentes poseen información fuertemente relacionada con el Dominio del Problema; los mismos han sido diseñados para interactuar con el usuario con el objetivo de resolver el problema para el cual el sistema ha sido desarrollado. Los componentes de la interfaz brindan información respecto a términos del Dominio del Problema, qué tipo de función cumple cada uno, la vinculación con las partes del código que implementan el mismo, clasificación de funcionalidades, entre otros aspectos.

Métricas de Software

Se agruparon distintos tipos de trabajos en el contexto de métricas de software y Comprensión de Programas. En cierta forma el uso de métricas ha formado parte de muchas estrategias de comprensión a lo largo de los años. Particularmente, en este capítulo se mencionan aquellos enfoques que pueden clasificarse en dos tipos: métricas para ser utilizadas como *índices de comprensibilidad* y métricas que infieren la importancia de un artefacto del código fuente en particular.

En la bibliografía se encontraron pocos enfoques que intenten inferir la importancia de un artefacto con respecto a la lógica subyacente del sistema. De hecho, casi la totalidad de los trabajos mencionados poseen dos desventajas en común:

- la mayoría ha sido diseñado para trabajar con granularidad a nivel de clases, lo cual hace que la propuesta sea difícil de contemplar en aproximaciones que analizan artefactos de granularidad más fina;
- ninguna de las propuestas usa un enfoque multicriterio, la mayoría adaptan algoritmos conocidos de evaluación y ordenamiento en conjunto con algún criterio estructural para inferir los puntajes.

Ambos aspectos son importantes ya que muchas de la propuestas de CPhan sido implementadas para trabajar con artefactos de software de granularidad más fina como métodos, estructuras de datos, uso de variables, etc. Pocos enfoques de evaluación y ordenamiento a nivel de clases son efectivos cuando se trabaja con artefactos de este tipo.

Por otra parte, es importante remarcar que los artefactos de software poseen distintas características que pueden calificarlo con respecto al sistema. Por ejemplo, en un sistema Orientado a Objetos, es importante conocer si un método es público o privado, ya que eso puede determinar la accesibilidad que el mismo posee con respecto al sistema en general. Claramente, si es privado, sólo será accesible desde la clase y eso indica que la funcionalidad del mismo está limitada al uso dentro de la clase. Siguiendo con el mismo ejemplo, sería importante conocer si el método es un manejador de un botón en la interfaz gráfica, ya que eso indicaría que dicho método está fuertemente ligado a la funcionalidad asociada al botón correspondiente. En este sentido, es difícil que estas características puedan ser reflejadas de manera íntegra si se utiliza una métrica que refleja sólo uno de los aspectos del artefacto. Por lo tanto, es necesario integrar distintas métricas que representen las diversas características de un artefacto para lograr reflejar completamente la importancia relativa del mismo en referencia al sistema completo.

3.3.1. Conclusiones

Con el fin de abordar los inconvenientes de las propuestas presentadas en el capítulo, en esta tesis doctoral se expone una estrategia que presenta las siguientes características:

- Usa representaciones internas las cuales son obtenidas exclusivamente a partir de artefactos del código fuente del sistema. De esta forma se evita el requerimiento de distintas fuentes de información y artefactos de gestión sofisticados (como por ejemplo, modelos del Dominio del Problema, información de equipos de desarrollo, etc.) que frecuentemente no están presentes en los proyectos de software que se quieren analizar (Koschke y Eisenbarth, 2000).

- Calcula una métrica que provee una estimación de la importancia relativa para cada artefacto de software analizado con respecto a la lógica subyacente del sistema. Dicha métrica se computa usando un método de evaluación multicriterio denominado LSP. Dicho método permite agregar los valores de distintas métricas mediante el uso de operadores lógicos. Esto permite obtener un valor que se verá afectado por la medición de los distintos aspectos que caracterizan a cada artefacto

analizado. Dicha métrica se usa para filtrar información irrelevante y para asistir al proceso de clustering.

- Propone una técnica de filtrado de información centrada en una métrica de importancia la cual toma diferentes aspectos de cada artefacto de software analizado.
- Define e implementa una técnica de clustering que toma como principal criterio de agrupamiento cierta información de los artefactos de software analizados y los *widgets* de la GUI del sistema, componentes estrechamente relacionados con el Dominio del Problema.
- Permite inferir un modelo de casos de uso que comprende: casos de uso, relaciones «*include*» entre casos de uso y *asociaciones* (relaciones entre el actor y los casos de uso). Esto en contraste con la mayoría de los trabajos relacionados los cuales no detectan las relaciones entre casos de uso.
- Implementa un algoritmo de clustering *cuasi-automático* con una leve asistencia del ingeniero de software para definición de rótulos, en contraste con muchos de los enfoques mencionados en el Apartado 3.2.2, los cuales centran sus enfoques en las decisiones del ingeniero que termina siendo determinante en la efectividad de las propuestas.

Capítulo 4

SAPSI: una estrategia para asistir a la Comprensión de Programas

> *La Computación es una ciencia de abstracción, es decir, crear el modelo correcto para un problema en particular y diseñar las técnicas mecanizables apropiadas para resolverlo.*
>
> — Alfred Aho

En este capítulo se presenta de manera concisa SAPSI, una estrategia que propone distintos tipos de análisis de la información de un sistema de software con el objetivo de vincular los Dominios del Problema y del Programa a partir de la obtención de un conjunto de casos de uso y sus dependencias. El principal objetivo de la estrategia es asistir al ingeniero de software durante la comprensión de un sistema de software particular. Un modelo de casos de uso permite examinar de manera sencilla el conjunto de funcionalidades existentes en el sistema bajo estudio, las dependencias entre las mismas y también posibilita analizar la interacción que existe entre los usuarios y el sistema. En este sentido, la estrategia plantea generar un modelo de casos de uso de manera *cuasi-automática* que además permita conocer aquellos artefactos de código fuente que implementan cada funcionalidad reflejada por los casos de uso obtenidos. En el contexto de CP, este tipo de enfoques permiten generar puentes cognitivos entre las abstracciones

del Dominio del Problema y los artefactos en el Dominio del Programa (Rugaber, 1995; Berón, 2010; Brooks, 1983; Von Mayrhauser y Vans, 1995).

En la Figura 4.1 se exhibe un modelo *Business Process Modeling and Notation* (BPMN) con las principales etapas que la estrategia lleva a cabo para obtener un modelo de caso de uso del sistema bajo estudio.

Es importante destacar que las actividades desarrollados en la primer etapa forman parte del *front-end* de la herramienta, mientras que las demás etapas integran el *back-end* (en el BPMN de la Figura 4.1 se puede visualizar este agrupamiento de procesos). La primer etapa está estrechamente vinculada con las tecnologías que se usaron para el desarrollo del sistema bajo estudio, como por ejemplo el paradigma de programación predominante en el sistema, el lenguaje de programación utilizado, la librería gráfica empleada para el desarrollo de la GUI, entre otros. Esta etapa permite obtener las representaciones básicas a través de las cuales la estrategia llevará a cabo su análisis. Una vez construidas estas representaciones, las demás etapas implementan el *back-end* de la herramienta, es decir, aquellos procesos que son independientes de un lenguaje o framework en particular Por lo tanto, cambiando el *front-end* es posible adaptar todo el proceso a distintos lenguajes de programación y librerías gráficas. De esta manera, la estrategia no está ligada a un único lenguaje de programación ni ciertas librerías gráficas, sino que es adaptable dentro del contexto de lenguajes multiparadigma de propósito general y las librerías gráficas que los mismos puedan utilizar. El único requerimiento es que todos los *front-ends* implementados generen las representaciones de base que SAPSI utiliza para llevar a cabo su objetivo. En este sentido, los procesos que conforman el *back-end* siempre llevarán a cabo las mismas actividades para cualquier sistema analizado.

En los próximos apartados se describen brevemente las actividades que cada una de estas etapas ejecuta y se muestran las desagregaciones de los procesos exhibidos en la Figura4.1.

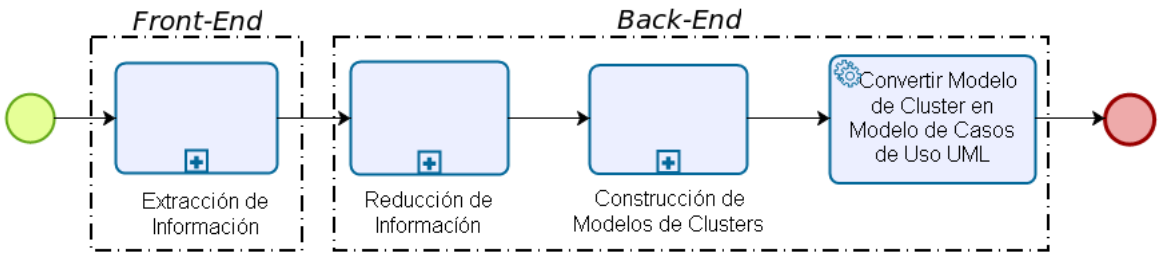

Figura 4.1: Modelo BPMN con las etapas de SAPSI.

4.1. Etapa 1: Extracción de la Información

Como se mencionó en el capítulo 2, en toda estrategia de CP, uno de los primeros pasos requiere la extracción de información del sistema bajo estudio. De acuerdo a la información que se intente analizar y al objetivo que se persigue, se debe optar por el tipo de técnica de extracción más conveniente. En caso en que la estrategia implementada proponga el análisis de información relacionada con tiempo de ejecución del sistema, será necesario utilizar TEI dinámica. En general esta técnicas se basan en la extracción de información generada durante la ejecución del sistema y su posterior análisis (Ball, 1999; Zaidman y cols., 2005; Cornelissen y cols., 2009). Este tipo de información posee ciertas ventajas y desventajas que se deben tener en cuenta a la hora de la estructuración de la estrategia. Una de las ventajas más esenciales es que la información recolectada contiene aspectos que efectivamente fueron utilizados por el sistema. Otros de las características relevantes de este tipo de información es que permite definir al usuario las funcionalidades que se desean estudiar. Además, como es de común conocimiento, existe información que sólo puede ser obtenida en tiempo de ejecución (por ejemplo, ligadura dinámica, valores de variables, excepciones, etc.). Sin embargo, una de las principales desventajas de analizar información dinámica es que para la ejecución del sistema es necesario contar con escenarios de ejecución que permitan destacar las funcionalidades que se desean analizar. Esto hace que la información recolectada, y en consecuencia el resultado de la estrategia, dependan de la precisión del ingeniero de software para generar dichos escenarios. Generalmente, esto requiere de un conocimiento determinado del sistema y del domino bajo estudio para generar los escenarios más relevantes de analizar. Por otra

parte, el tamaño de las trazas de ejecución suele ser considerable y en determinados casos se torna inmanejable.

En contraposición a las desventajas que trae aparejada la extracción y análisis de información dinámica, la información estática puede ser analizada de manera independiente de la ejecución del sistema. Por lo general, las TEI estática utilizan como base artefactos del código fuente del sistema bajo análisis. Otro de los aspectos positivos de la información estática de un sistema es que la misma está limitada y puede ser completamente estimada a partir del código fuente.

Es por los motivos precedentes que SAPSI utiliza e implementa TEI estáticas para extraer la información del sistema bajo estudio, la cual tomará como base para llevar a cabo todo el proceso planteado.

En este paso en particular, la estrategia extrae información relativa a distintos aspectos del código fuente que son reflejados y administrados en la siguientes estructuras (las mismas serán detalladas en el Capítulo 5):

- un grafo estático de llamadas a funciones (integra funciones y métodos) ,
- una representación jerárquica de la GUI del sistema donde figuran los *widgets* [1] utilizados, y
- un conjunto de métricas de software.

El grafo de llamadas a funciones será la estructura de base que se construye para representar la información extraída. El mismo permite trabajar con granularidad a nivel de métodos o funciones. De manera sintética, la representación jerárquica de la interfaz gráfica del sistema y las métricas se utilizan para aumentar la cantidad de información de los elementos en el grafo. De esta manera se hace referencia a un grafo "atribuido" (o aumentado) con la información de la GUI y las métricas extraídas.

Toda la información recolectada y representada en esta etapa es integrada y posteriormente utilizada durante todo el proceso que comprende la estrategia.

En la Figura 4.2 se puede visualizar un modelo BPMN con las principales actividades llevadas a cabo en esta etapa.

[1] Los *widgets* son componentes que proveen las librerías gráficas para diseño de las GUI a través de los cuales el usuario puede interactuar con el sistema (por ej. ventanas, botones, listas desplegables, campos de texto, etc.).

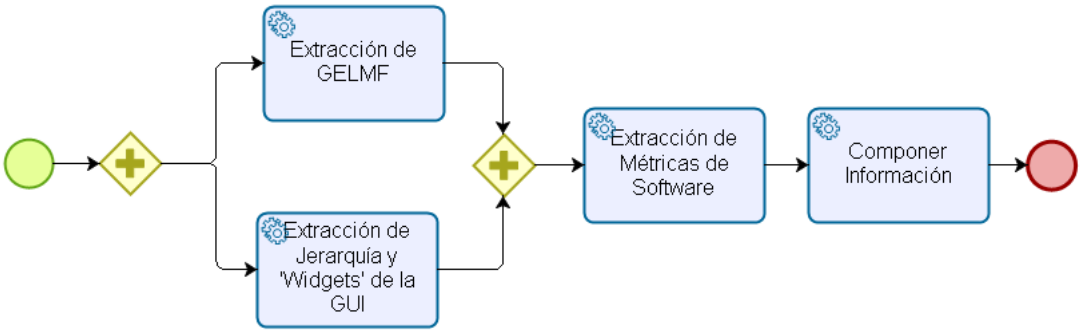

Figura 4.2: Subprocesos de la actividad Extracción de Información.

En el Capítulo 5 se detallan los conceptos, técnicas, estructuras y herramientas relativos a desarrollo de esta etapa.

4.2. Etapa 2: Reducción de Información

En el contexto de análisis de software de gran envergadura, es necesario contar con técnicas de reducción de información para poder generar abstracciones. Por lo general, en dichos sistemas se extrae cierta cantidad de información que no está vinculada con aspectos relativos a la solución del problema para el cual fue desarrollado el sistema. Como claro ejemplo de esto se pueden considerar las inicializaciones de variables, uso de librerías externas, configuraciones relativas al lenguaje de programación utilizado, etc.

Para que las representaciones de base contengan información precisa referente al problema que el sistema intenta solucionar, es necesario filtrar aquellos artefactos de software que no están relacionados con la lógica subyacente del programa. En este contexto, SAPSI lleva a cabo un proceso de filtrado de información irrelevante encontrada en las representaciones de base construidas en la etapa previa. En este sentido se calculan ciertas métricas de software que en conjunto permiten obtener una única métrica que representa la importancia relativa de cada artefacto de software considerado en el grafo construido en referencia al sistema bajo estudio. Para esto es necesario utilizar cierto tipo de operaciones y así lograr converger, en cierta manera, los valores obtenidos de las métricas consideradas en un único valor que indique un índice de importancia para cada artefacto de software analizado. Con este propósito, se utiliza un método de eva-

luación multicriterio denominado LSP. Este método provee un conjunto de operadores lógicos que posibilitan agregar valores en pos de obtener estimaciones que representan satisfacciones de las distintas métricas agregadas.

La estrategia implementa el método de evaluación antes mencionado integrando los siguientes aspectos:

- utiliza a las métricas extraídas en la etapa previa como el conjunto de criterios que el método toma como base,
- define funciones de normalización para los rangos dispares de valores que poseen las distintas métricas,
- especifica una estructura de agregación compuesta por operadores lógicos que posibilitan la integración de los valores de las métricas en un único valor que proveerá un índice de importancia para cada artefacto considerado en el grafo.
- calcula el índice de importancia para cada artefacto que compone el grafo de llamadas a funciones atribuido.

Como se muestra en el BPMN de la Figura 4.3, el filtrado es iterativo hasta que todos nodos del Grafo Estático de Llamadas a Métodos/Funciones (GELMF) superen un umbral previamente definidos por el usuario. Por cada iteración, se realiza un reajuste de la métrica de importancia previamente calculada.

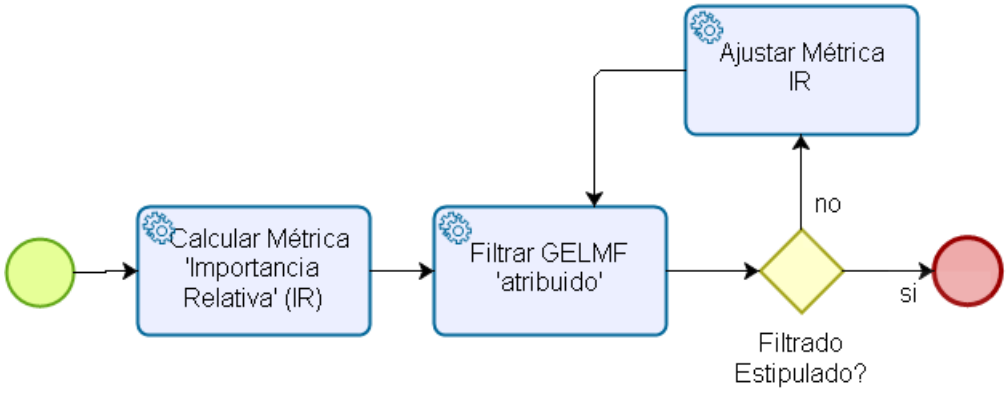

Figura 4.3: Subprocesos de la actividad Reducción de Información.

El Capítulo 6 presenta todos los conceptos y modelos implicados en el cálculo de esta métrica y el algoritmo de reducción de información.

4.3. Etapa 3: Construcción de Clusters

Si bien el filtrado de información irrelevante ayuda a reducir el grafo de llamadas a funciones, para cualquier sistema de gran tamaño se requerirá además un mecanismo de abstracción de la información. Sin ir más lejos, en un sistema relativamente pequeño la representación puede alcanzar un volumen difícil de manejar para el ingeniero de software. Por ejemplo, para tener noción del tamaño que puede alcanzar la representación, un sistema de aproximadamente 10 KLOCs en lenguaje puede generar un grafo de aproximadamente 2400 nodos y 8600 arcos. Un filtrado básico aplicado a este grafo resulta en una estructura con 730 nodos y 540 arcos (Miranda y cols., 2015). Claramente, esto va a depender del lenguaje de programación, el tipo de sistema y el entorno en donde se ejecuta. Sin embargo, aunque la reducción de información fue significativa, igualmente la representación sigue siendo difícil de visualizar e inmanejable en referencia a las tareas que deberá realizar el ingeniero de software para comprender el sistema (demasiados nodos y arcos) .

En este sentido, es posible caracterizar a las técnicas de clustering como unos de los mecanismos más utilizados en el contexto de Ingeniería Reversa para agrupar y abstraer artefactos de un sistema de software. Las técnicas de clustering permiten agrupar entidades de un sistema, tales como clases, funciones o archivos, en componentes significativas (por ejemplo, subsistemas) con el objetivo de asistir en la comprensión de la estructura de un sistema de gran envergadura.

En esta etapa la estrategia implementa una técnica de clustering la cual es primordial con respecto a la abstracción e identificación de las funcionalidades elementales del sistema bajo estudio. Con este propósito la estrategia define un técnica de clustering aglomerativa que va agrupando las entidades del grafo de llamadas a funciones formado de esta manera un modelo de cluster del sistema. Para esto se toma como base información extraída en etapas previas. Es por este motivo que, como se puede visualizar en la Figura 4.1, la técnica de clustering es iterativa hasta que todos los elementos del grafo de llamadas son agrupados. Esta información refleja distintos aspectos del sistema analizando tales como los componentes de la interfaz gráfica de usuario, las relaciones en el grafo de llamadas a funciones entre entidades de software consideradas, el rol to-

pológico que cumple cada artefacto en grafo de llamadas, entre otros. Sin embargo, el criterio principal que dirige el algoritmo de clustering propuesto está relacionado con los componentes de la interfaz gráfica de usuario. Estos cumplen un rol fundamental en el sistema y están estrechamente relacionados con el Dominio del Problema (Memon y cols., 2003; Almendros-Jimenez y Iribarne, 2005). El algoritmo realiza un proceso iterativo donde aquellos artefactos de software que son más relevantes para la lógica subyacente del sistema, comienzan a "absorber" los demás definiendo los clusters más importantes del sistema.

El algoritmo planteado sigue un lineamiento similar al que introducen Tzerpos y Holt (Tzerpos y Holt, 2000), donde los autores afirman que, en el contexto de CP, el algoritmo de clustering debería apuntar a identificar ciertos patrones para agrupar entidades. En el trabajo se plantea que la mayoría de los algoritmos de clustering de software proponen maximizar una métrica (Mancoridis y cols., 1998, 1999) o agrupar elementos de acuerdo a una medida de similaridad basada en información limitada a ciertos aspectos del sistema (Erdemir y cols., 2011; Maqbool y Babri, 2004). Sin embargo, este tipo de enfoques suelen perder de vista el principal objetivo de clustering de software: facilitar el entendimiento de los sistemas a los ingenieros de software.

El algoritmo de clustering propuesto en el marco de SAPSI, agrupa las entidades teniendo como base ciertos patrones y heurísticas de agrupamiento que fomentan el descubrimiento de componentes de más alto nivel con fuerte significado semántico sobre el sistema. El objetivo principal del algoritmo es obtener un modelo de clusters que sea mapeable con un modelo de casos de uso del sistema. Una vez que se ha construido el modelo de clusters, es decir el conjunto de clusters y las relaciones entre los mismos, el próximo paso es generar un modelo de casos de uso basado en el de clusters.

Es importante destacar que la estrategia facilita el rotulado de los clusters tomando como base términos relativos a los componentes de la GUI del sistema. Como se mencionó previamente, la interfaz gráfica del sistema tiene embebidos muchos términos del Dominio del Problema que pueden ser usados para describir de manera efectiva las funcionalidades integradas en los clusters. Esta es una característica importante, ya que una estrategia de nombramiento débil puede hacer fracasar toda la estrategia de comprensión.

En el BPMN de la Figura 4.4 se muestra las actividades relacionadas con este proceso.

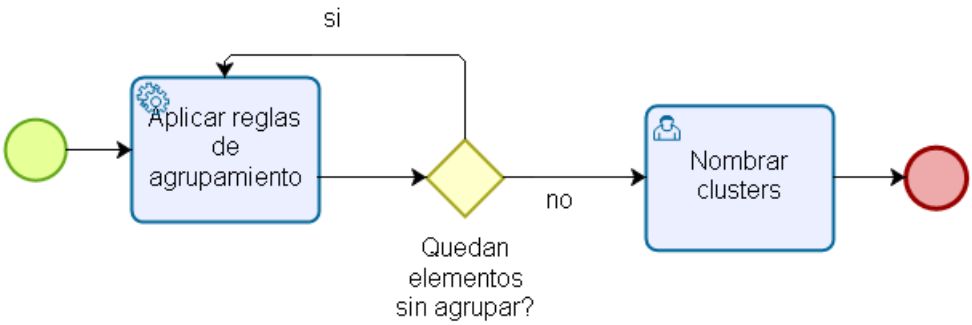

Figura 4.4: Subprocesos de la actividad Construcción de Modelo de Clusters

Los detalles de los procesos involucrados en la técnica de clustering propuesta son explicadas en el Capítulo 7.

4.4. Etapa 4: Obtención de Casos de Uso

Un caso de uso es un modelo para describir las funcionalidades del sistema y para representar las interacción que existen entre el sistema y aquello que lo "rodea" (Booch y cols., 2004). Dirigir el proceso de desarrollo, reingeniería y/o Ingeniería Reversa basado en casos de uso posee ciertas ventajas (Jarke, 1999; Seffah y cols., 2001):

- los casos de uso son fáciles de entender para los involucrados en el proceso de desarrollo de software. Existen varios factores que facilitan su entendimiento, entre los que se pueden mencionar:
  - los elementos del diagrama de casos de uso son pocos, y sus funciones no son complejas de comprender.
  - la naturaleza del modelo es simple ya que representan funcionalidades del sistema y las interacciones del mismo con su entorno. Es decir, no representan actividades o tareas de bajo nivel.
  - es un modelo que está fuertemente ligado al Dominio del Problema, por lo tanto contiene mucha terminología del mismo.
- los modelos de casos de uso pueden ser usados para generar casos de prueba (Escalona y cols., 2011; Dunsmore y cols., 2003), ya que describen el comportamiento externo

del sistema desde el punto de vista del usuario.

- vincular los artefactos de software del código fuente con los casos de uso facilita la aplicación de *Técnicas de Análisis de Impacto* [2] (Lindvall y Sandahl, 1998).

- el modelo de caso de uso extraído puede ser utilizado para verificar de manera automática si el sistema cumple con los requerimientos funcionales especificados por el cliente (Couto y cols., 2014).

Por lo tanto, la obtención de casos de uso en un contexto de CP resulta por demás interesante ya que el ingeniero de software podrá vincular aquellos artefactos del Dominio del Programa que implementan las funcionalidades del sistema especificadas en el modelo de casos de uso.

Es por este motivo que en la última etapa, la estrategia mapea el modelo de cluster generado en la etapa previa en un modelo de casos de uso de UML. Dicho modelo de casos de usos no solo brinda una vista que refleja las funcionalidades del sistema y las dependencias entra las mismas (casos de uso y las relaciones << *include* >>), sino que además posibilita la inspección de aquellos artefactos del código fuente que están estrechamente vinculados con cada caso de uso particular.

En el modelo BPMN de la Figura 4.1 se puede observar que esta última actividad es "atómica", a diferencia de las restantes que pueden ser desagregadas en subprocesos. El capítulo 7 detalla específicamente el proceso de mapeo descripto en este apartado.

4.5. Conclusiones

En esta tesis doctoral se propone una estrategia denominada SAPSI que asiste al ingeniero de software durante el proceso de comprensión de un sistema particular. La misma tiene como principal objetivo derivar un modelo de casos de uso de UML para el sistema bajo análisis. Para esto define la siguiente serie de pasos:

1. Extrae información del sistema haciendo uso de TEI estática que analizan el código fuente para derivar: i) un grafo de llamada a funciones, ii) una estructura que

[2] El Análisis de Impacto de cambios en un sistema tiene como propósito prever los efectos que tendrá un cambio propuesto, identificando qué elementos se verán afectados y de qué forma.

representa la GUI del sistema y la jerarquía de sus componentes y iii) métricas de software.

2. Ejecuta un método de reducción de información basada en la información extraída en el paso previo.

3. Obtiene un modelo de clusters mediante el uso de una técnica de clustering principalmente basada en los componentes de la GUI del sistema.

4. Deriva un modelo de casos de uso partiendo del modelo de cluster obtenido en el paso previo. Este modelo integra los componentes del código fuente ligados a cada funcionalidad descripta en el mismo.

Trabajar con modelos de casos de uso tiene ciertas ventajas en el contexto de CP ya que, entre otras cosas, los mismos son fáciles de comprender por los involucrados en el proceso de desarrollo de software, reingeniería e Ingeniería Reversa (Bojic y Velasevic, 2000; L. Zhang y cols., 2006; Budgen y cols., 2011). Además, la estrategia también permite la visualización de las dependencias entre elementos del modelo de casos de uso y los artefactos del código fuente relacionados. Este tipo de enfoques son valiosos en el contexto de CP ya que facilitan los procesos de vinculación entre los artefactos del Dominio del Programa y los componentes del Dominio del Problema.

En los próximos capítulos se describirán más detalladamente los conceptos y procesos llevados a cabo en las distintas etapas de la estrategia planteada. Por otra parte, se presenta una herramienta que implementa la estrategia descripta en conjunto con casos de estudio que muestran la aplicabilidad del enfoque.

Capítulo 5

Extracción de Información

La verdad únicamente se puede encontrar un sólo lugar: el código fuente.

— Robert C. Martin

La extracción de información es una disciplina que se enfoca en la obtención de información relevante de un sistema de software a partir de los distintos "recursos" que el mismo puede proveer. De acuerdo a los aspectos que se quieran inspeccionar y al tipo de información disponible del sistema, es que se pueden utilizar principalmente TEI dinámica o estática.

Como se introdujo en el capítulo precedente, SAPSI utiliza TEI estática para obtener información acerca del sistema bajo estudio. La misma está relacionada con llamadas entre métodos y/o funciones en el código fuente y la conexión de los mismos con los componentes de la GUI del sistema bajo análisis. A diferencia de otras disciplinas que adoptan ciertos tipos de estructuras limitadas para representar la información extraída[1], el código fuente del sistema provee diversos aspectos que se pueden utilizar para reforzar la información recolectada en una representación de base. En este caso en particular SAPSI "atribuye" un Grafo de Llamadas a Funciones con métricas de software específicas que permiten analizar con mayor profundidad los artefactos extraídos y la interrelación

[1]Por ejemplo, en la mayoría de los problemas en biología las redes se inspeccionan tomando como base exclusivamente las relaciones entre nodos, por lo tanto, la mayoría de resultados obtenidos provienen del análisis de la topología de la red.

entre los mismos.

El capítulo se organiza de la siguiente manera, en primer lugar se introducen las TEI más utilizadas en el contexto de CP (Sección 5.1); posteriormente se describen la información obtenida y las representaciones construidas por SAPSI en conjunto con los procesos y conceptos implicados en la extracción de las mismas (Sección 5.2); finalmente se presentan las notas y comentarios del capítulo (Sección 5.3).

5.1. Técnicas de Extracción de Información Estática

Si bien existe un gran número de técnicas que permiten obtener información a partir de recursos de diversa índole, en esta sección se exponen las TEI estática más utilizadas en el contexto de CP.

5.1.1. Técnicas *ad-hoc*

El uso de estas técnicas en el contexto de CP se basa en el empleo de comandos del sistema operativo orientados a la extracción de información desde distintos tipos de recursos. Estos comandos se pueden concebir como programas que proveen los sistemas operativos y que pueden ser invocados desde algun intérprete de comandos. Los mismos pueden ser integrados en conjunto mediante srcipts desarrollados por el usario[2].

Los comandos provistos por los sistemas operativos son utilizados con numerosos propósitos, específicamente este apartado menciona un conjunto de comandos de Unix que posibilitan la obtención de información específica (Bach y cols., 1986). La misma se puede utilizar con propósitos de inspección de los diversos aspectos de bajo nivel de un sistema, tales como los archivos fuentes, datos del sistema operativo, información de ejecución de un proceso en particular, etc.

- *grep:* sirve para mostrar las líneas de un fichero que contienen un patrón, es decir que contienen un texto. Si se especifican varios ficheros, se buscará en todos ellos; si no se especifica ninguno, se buscará en la entrada estándar. El funcionamiento

[2]Los procesos y conceptos que rodean a los comandos provistos por sistemas operativos están fuera del alcance de esta investigación. Si el lector está interesado en profundizar conceptos referidos a estas temáticas puede leer la bibliografía pertinente (Sobell y Helmke, 2005).

normal del comando es mostrar en la salida estándar las líneas que contienen el patrón buscado.

- *sed:* La orden sed es un filtro como *grep*, pero permite hacer modificaciones a los ficheros. *sed* lee su entrada línea a línea y escribe las líneas una a una en su salida estándar. Por cada línea leída, *sed* aplica una operación de sustitución de la forma utilizada con *ed*. Si se produce la coincidencia, se realiza la sustitución y se escribe la línea; si no hay coincidencia, la línea se escribe sin modificar. Dicho comando puede ser usado para realizar transformaciones de código en el contexto de Ingeniería Reversa. El comando *grep* es una aproximación simple y muy útil para recuperar diversos aspectos sintácticos del código fuente del sistema. El comando se puede usar para el análisis de información tanto formal (funciones, declaraciones, sentencias, etc.) como informal (documentación en el código, comentarios, etc.)

- *nm:* se usa como ayuda en la depuración al examinar ficheros binarios, incluyendo bibliotecas, módulos objeto y ejecutables. Su salida muestra la tabla de símbolos, concretamente una línea por cada símbolo que se encuentre en la biblioteca o archivo objeto especificado. De cada símbolo especifica su tamaño en bytes, el tipo de objeto, su ámbito y su nombre. Un ejemplo del uso de este comando es en contextos donde no se cuenta con el código fuente del sistema y se deben analizar los archivos ejecutables.

- *awk:* explora una lista de ficheros de entrada buscando líneas que se correspondan con un conjunto de patrones especificados. Por cada patrón que coincida, se efectúa un conjunto especificado de acciones. Estas acciones pueden implicar manipulaciones de campos dentro de la línea u operaciones aritméticas sobre los valores de los campos. La herramienta *awk* es un lenguaje de programación con características de los lenguajes de programación de propósito general tales como shell, bc y C. Por este motivo suele ser ampliamente utilizado en contextos de Ingeniería Reversa a bajo nivel, por ejemplo, en situaciones donde se deben analizar y utilizar aspectos vinculados al sistema operativo.

- *diff:* produce un índice completo de todas las líneas que difieren entre dos archivos,

junto con sus números de líneas y lo que debe ser modificado para hacer iguales los ficheros. Es un comando básico que puede ser usado con múltiples propósitos en el contexto de Ingeniería Reversa, por ejemplo, para determinar duplicación de código en el proyecto analizado.

- *objdump:* es utilizado para extraer información a partir de archivos objeto. A diferencia de los anteriores, el propósito de este comando siempre se enmarca en tareas ligadas a desarrollo e inspección de software, es por este motivo que el mismo es ampliamente utilizado en entornos relativos a Ingeniería de Software.

Es importante remarcar que este tipo de técnicas poseen ciertas ventajas a la hora de buscar concordancias sintácticas. Sin embargo, es difícil mantener el contexto de la estructura general del sistema en donde se realiza la búsqueda. Por ejemplo, no es sencillo responder a las siguientes preguntas ¿En qué función del código se encontró la cadena buscada? ¿Qué funciones poseen la cadena buscada dentro de sus correspondientes documentaciones?

5.1.2. Técnicas dirigidas por la Sintaxis

Una de las TEI más convencionales en el contexto de CP es mediante el uso de gramáticas de atributos y definiciones dirigidas por la sintaxis (A. V. Aho y cols., 2007).

Por lo general, el uso de este tipo de técnicas conlleva un proceso el cual se puede ver reflejado en el BPMN exhibido la Figura 5.1. En los siguientes apartados se describe brevemente las actividades asociadas a cada etapa representada y las representaciones generadas por las mismas.

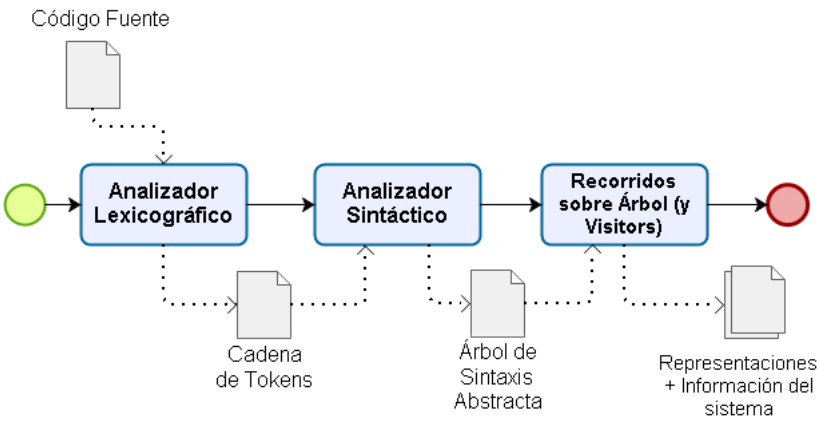

Figura 5.1: Extracción de información usando gramáticas.

Analizador Lexicográfico

El *analizador lexicográfico*, *lexer* o *scanner*, es el primer paso en el proceso completo, por lo tanto es el que recibe el código fuente del sistema. El programa es recibido como una cadena de entrada y lexer la divide en unidades lexicográficas llamadas *tokens*. Cada *token* es una secuencia de caracteres que representa una unidad de información en el programa fuente, por ejemplo, cualquier tipo de operadores, identificadores, palabras reservadas, etc.

En el Algoritmo 1, se muestra un ejemplo sencillo del uso de un lexer para comprobar si las llaves que abren se corresponden con las llaves que cierran en un programa escrito en Java o en C/C++. La idea es simple, no obstante la implementación de la función *obtener_token*(), que realiza el *tokenizer* (es decir divide el texto de un programa en tokens), no es sencilla ya que consume mucho tiempo desarrollarla y puede ser propensa a errores. Para evitar este tipo de inconveniente, es aconsejable usar herramientas que faciliten esta tarea. Algunas de las herramientas más utilizadas con estos propósitos son los generadores de lexers. Las mismas reciben una especificación orientada a comparar literales de caracteres, y produce un programa que reconoce expresiones regulares. Una vez que se le pasa una cadena para ser analizada, el lexer generado reconoce estas expresiones y las divide en cadenas de caracteres que coinciden con las expresiones especificadas. Los generadores más conocidos son *Lex*, *AFLEX*, *JFlex*, entre otros.

```
Input: cadenaEntrada // cadena a analizar
Result: verificacion = True // resultado del analisis
Data: pila // pila para controlar llaves
1  pila.inicilizar();
2  while token=obtener_token() != Null do
3  |  if token='{' then
4  |  |  pila.push('{');
5  |  end
6  |  else if token='}' then
7  |  |  if pila.isEmpty() then
8  |  |  |  verificacion=False;
9  |  |  |  break;
10 |  |  end
11 |  |  else
12 |  |  |  pila.pop()
13 |  |  end
14 |  end
15 end
16 if !pila.isEmpty() then
17 |  verificacion=False;
18 end
```

Algoritmo 1: Verificador de llaves

Analizador Sintáctico

La segunda fase del proceso es el análisis sintáctico o *parsing*. El parser usa los *tokens* que genera el analizador lexicográfico para crear una estructura intermedia con forma de árbol que representa la estructura gramatical del conjunto de *tokens* recibidos. Una representación ampliamente utilizada con estos propósitos son los árboles de sintaxis donde cada nodo interior representan una operación y los hijos del nodo representan los argumentos de dicha operación. La principal función de un analizador sintáctico es determinar si una frase gramatical está sintáctcamente bien escrita.

La construcción de un analizador sintáctico se basa sobre la definición de una gramática que indica como se escriben las sentencias correctamente.

Una gramática es una cuádrupla $G = (N, T, P, C)$ donde:

- N: es el conjunto de símbolos no terminales. Estos representan las construcciones sintácticas imponiendo una estructura jerárquica sobre el lenguaje la cual es fundamental para la traducción dirigida por la sintaxis.

- T: es el conjunto de símbolos terminales. Es decir, los símbolos básicos que forman las cadenas de entrada. Por ejemplo las palabras reservadas del lenguaje, tales como *if*, *else*, *while*, etc.
- P: es un conjunto de producciones que permiten determinar cómo pueden ser combinadas para formar cadenas. Una producción consiste en un encabezado, el cual representa el no terminal que será reemplazado, y un cuerpo, la cadena compuesta de símbolos gramaticales que reemplazará al encabezado.
- C: es el símbolo de comienzo.

En base a las reglas gramaticales, el parser construyendo una estructura arbolada donde cada nodo interno representa un no terminal y cada hoja un nodo terminal. De esta manera se representa el programa usando una estructura que facilita la búsqueda de información del sistema, la misma se denomina *Árbol de Parsing*. El lector interesado en los distintos conceptos y variantes que presenta el análisis sintáctico, puede consultar la bibliografía referente a la temática (A. V. Aho y cols., 2007; Kenneth, 1997) .

De la misma forma que los *lexers*, en la actualidad existen diversas herramientas para construir *parsers*, algunos ejemplos de las mismas son AntLR (Parr, 2013), YACC (Johnson, 1975) y LISA (Mernik y cols., 2002).

Árbol de Sintaxis Abstracta

Un *Árbol de Parsing* (o *Árbol de Sintaxis Concreto*) es una estructura que representa la estructura sintáctica de una cadena de caracteres de acuerdo a cierta gramática formal. Este representa toda la estructura de la cadena, es decir, representa toda la cadena en forma de árbol.

Un *Abstract Sintax Tree* (AST), es una representación de árbol de la estructura sintáctica simplificada del código fuente escrito en cierto lenguaje de programación. Cada nodo del árbol denota una construcción que ocurre en el código fuente. La sintaxis es abstracta en el sentido que no representa cada detalle que aparezca en la sintaxis verdadera. Es decir, en el mismo no figuran ciertos elementos irrelevantes como los delimitadores del lenguaje (llaves, punto y coma, coma, etc). Por ejemplo, el agrupamiento de los paréntesis está implícito en la estructura arborescente, y una construcción sintáctica

tal como $IF < condicion > THEN$ puede ser denotada por un solo nodo con dos ramas. En la Figura 5.2 se exhibe un ejemplo donde partiendo de la base de una gramática en particular, se exhiben ambos tipos de árboles para una cadena específica.

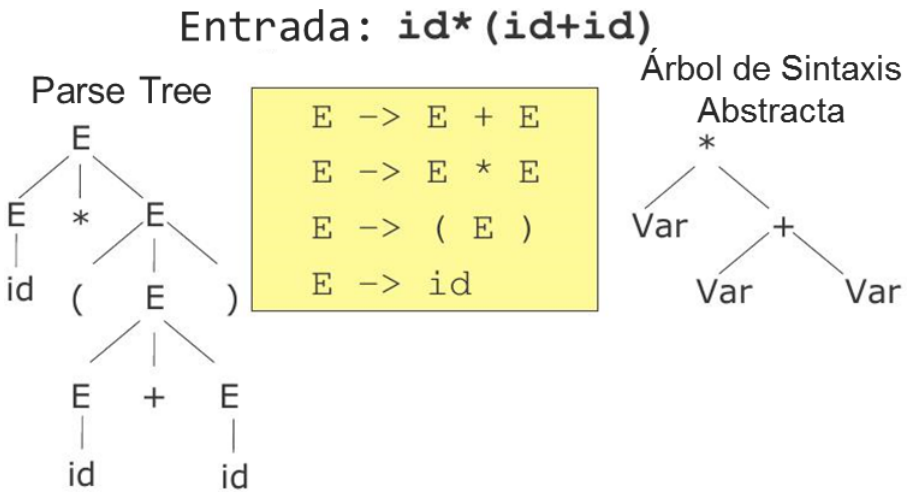

Figura 5.2: Diferencia entre AST y Árbol de Parsing.

Como es posible deducir de los párrafos anteriores, el análisis de un *Árbol de Parsing* para un sistema industrial puede llegar a ser tedioso debido al tamaño que el mismo puede alcanzar. En este sentido, el AST contiene las partes más sustanciales del código fuente, por lo tanto posee menor altura y contiene menos elementos. Además, una de las ventajas más interesantes que tiene trabajar con ASTs es que los mismos pueden ser editados y "atribuidos" con cierta información como propiedades y anotaciones para cada elemento que compone el mismo. Esto sería imposible de realizar usando sólo el código fuente ya que implicaría efectuar alteraciones al mismo.

Análisis de AST

Una vez que se ha generado el AST es necesario explorar el mismo para obtener información del sistema. A continuación se mencionan dos de los aspectos más relevantes relativos al análisis de AST:

Recorridos: existen diversos algoritmos de recorridos de árboles (o *traversals* por su denominación en inglés) que permiten obtener distintos aspectos de la información contenida AST (A. V. Aho y cols., 2007). El recorrido determina el orden en el

que se van "visitando" los nodos del AST. Como claros ejemplos de los mismos se pueden mencionar *post-order*, *pre-order*, *in-order*.

Visitors: si para la implementación del AST se usa un lenguaje orientado a objetos, entonces se puede hacer uso de polimorfismo para facilitar el recorrido de los diferentes nodo. Específicamente se utiliza el patrón de diseño *Visitor*, el cual posibilita separar el algoritmo de la estructura que se recorre.

En este contexto, los nodos en el AST se implementan normalmente con una variedad de clases, todas derivadas de una clase *NodoAST* común. Para cada construcción sintáctica en el lenguaje que se está procesando, habrá una clase para representar esa construcción en el AST, por ejemplo *NodoVariable* para nombres de variables, *NodoAsignacion* para operaciones de asignación del lenguaje, etc. Cada una de estas clases provee toda la información referente al tipo de nodo que está representado.

Cuando se recorre el AST, el uso de polimorfismo facilita el análisis de cada elemento ya que el nodo base *NodoAST*, define las operaciones básicas que se pueden realizar con cada nodo. Además, cada tipo de nodo particular implementa estas operaciones de acuerdo a la manera que define la construcción específica del lenguaje.

Una de las facilidades que proporciona implementar el AST usando este patrón es la capacidad de extraer la información requerida, pudiendo agregar nuevas operaciones sin modificar las estructura ni las operaciones ya existentes.

En esta sección se presentaron algunos de los conceptos más relevantes relativos a extracción de información estática. El lector interesado en profundizar más las temáticas relativas a esta disciplina, puede consultar la bibliografía más citada en este contexto (A. V. Aho y cols., 2007; Kenneth, 1997). En la siguiente sección se describen las técnicas y procesos utilizados por la estrategia para extraer información del sistema bajo estudio y de esta manera construir las representaciones de base usadas durante todo el proceso de análisis de SAPSI.

5.2. Representaciones Construidas por SAPSI

SAPSI utiliza TEI estática para obtener información del sistema bajo estudio mediante la construcción y análisis del AST construido a partir del programa. Específicamente, la estrategia extrae dos representaciones:

- GELMF: una clase de grafo de llamadas a funciones que posibilita representar programas escritos en lenguajes multiparadigma y cuyos nodos están atribuidos con información específica de cada método/función.
- Árbol de Representación de la GUI (ARGUI): contiene información de la jerarquía de la GUI del sistema y sus componentes (*widgets*).

Es de común conocimiento que este tipo de representaciones poseen un fuerte trasfondo matemático; facilitan la abstracción y representación y proveen numerosos algoritmos para operar con las mismas (Jungnickel, 2007). Estos factores se tornan interesantes a la hora de llevar a cabo distintos tipos de análisis ya que facilitan la administración y visualización de las estructuras.

Como se destacó en el capítulo 4, la etapa de extracción de información es el *front-end* de toda la estrategia, es decir los procesos desarrollados en este capítulo están relacionados con aspectos de bajo nivel empleados por el sistema bajo análisis tales como los lenguajes de programación usados, la sintaxis específica de los mismos, las librerías gráficas utilizados, etc. Las etapas restantes de la estrategia conforman el *back-end*, en otras palabras, aquellos procesos que llevan a cabo el análisis planteado por SAPSI, los cuales además no dependen de los detalles tecnológicos empleados por el sistema bajo estudio. En este sentido, por cada uno de los lenguajes y librerías gráficas utilizadas, es necesario definir y agregar un nuevo *front-end* a la estrategia, el cual extraiga la información y genere las representaciones de base usadas por SAPSI. De esta manera, la estrategia se concibe independiente del lenguaje de programación y la librerías gráficas utilizadas.

En los próximos apartados de esta subsección, se explican en detalle los procesos llevados a cabo en el *front-end* de la estrategia (Figura 5.3), específicamente, las representaciones de base que se construyen, las técnicas y procesos usados en la construcción

de las mismas, y la extracción de un conjunto de métricas que servirá para atribuir las representaciones generadas. En la Figura 5.4 se puede consultar un modelo BPMN con las actividades más importantes de la etpa.

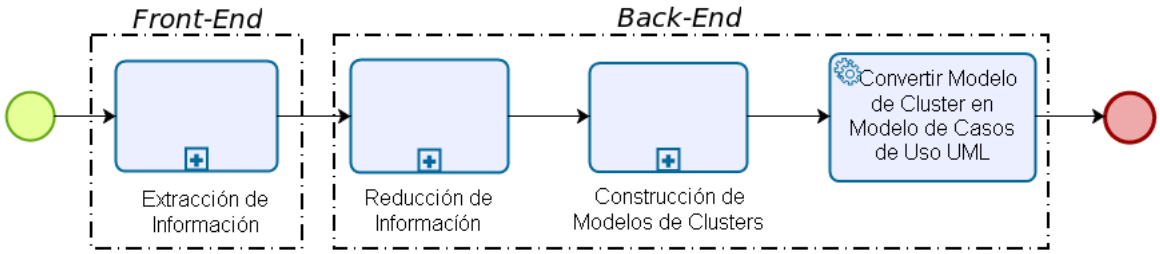

Figura 5.3: Modelo BPMN con las etapas de SAPSI agrupadas en procesos front-end y back-end.

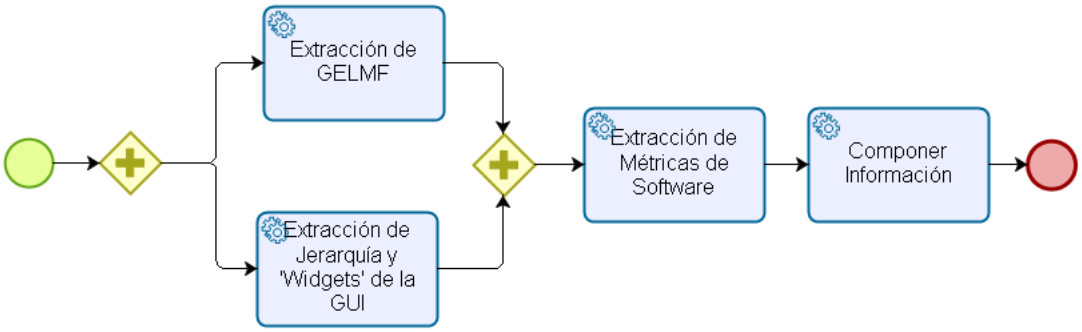

Figura 5.4: Subprocesos de la actividad Extracción de Información descriptos a lo largo del capítulo.

5.2.1. Grafo Estático de Llamadas a Métodos/Funciones

Una de las estructuras más utilizadas con propósitos de representación y análisis en el contexto de extracción de información estática es el *Grafo de Llamadas a Funciones* (o por sus siglas en inglés, *FCG*) (Eisenbarth y cols., 2001; Ali y Lhoták, 2012). La misma contiene las relaciones de llamadas entre funciones del sistema bajo análisis representando todos los escenarios de ejecución posibles. Para el caso de lenguajes multiparadigma, como es caso de Python, se ha redefinido el FCG para que incluya llamadas a funciones y métodos. Para estos casos en particular la estructura se definirá como GELMF.

Un GELMF es un grafo dirigido $G = (V, E)$ donde cada nodo en V representa un método o una función declarada en el código fuente del sistema, y cada arco $E = (v, u)$,

donde $v, u \in V$, representa una llamada desde el método/función v al método/función u. Esto muestra básicamente las relaciones de llamadas entre las "rutinas" del programa. Cada método/función (llamador) en el grafo puede tener un conjunto de posibles llamados, es decir aquellos que son invocados dentro del cuerpo del método/función llamador (si no hay invocaciones, este puede ser cero). En pocas palabras, en un GELMF de un sistema determinado, cada nodo representa un método o función en el código fuente, y un arco desde un nodo v a u se interpreta como una dependencia estática desde el método/función v a u. Cada nodo que representa un método se identifica con el nombre del mismo incluyendo el nombre de la clase y el archivo mientras que las funciones sólo se identifican con nombre y archivo. Además, cada nodo en el grafo ha sido *atribuido*[3] con un conjunto de métricas de software detalladas en la Subsección 5.2.6. Las mismas son empleadas con diferentes propósitos en la estrategia, en particular, para calcular la IR de cada nodo en relación a todo el sistema (ver Capítulo 6). Este tipo de representaciones es utilizada por muchos investigadores en el contexto de Ingeniería Reversa. Entre las tantas características que dicha representación provee, es posible resaltar las siguientes como las más esenciales:

- el proceso de construcción no presenta una gran complejidad y existen muchas herramientas de software que pueden asistir al mismo.
- el despliegue visual de esta representación es simple, fácil de comprender y muy versátil.
- la estructura posee un fuerte y sólido trasfondo matemático.
- existe un gran número de operaciones de grafos que permiten llevar a cabo distintos tipos de análisis sobre los mismos.

5.2.2. Construcción del GELMF

Para la construcción del GELMF se genera el AST del programa mediante el uso de técnicas de parsing y posteriormente, un módulo que implementa el patrón *visitor* (de

[3]El término *atribuido* se usa para denotar que a la información inherente al nodo en el GELMF (por ejemplo, nombre del archivo del método/función) se le adiciona nueva información que excede a la definición de GELMF.

ahora en adelante el "visitor") recorre el AST en busca de información específica acerca del código fuente.

En términos generales, el visitor recorre el AST en busca de aquellos nodos que representen las siguientes construcciones del lenguaje:

- Métodos/Funciones

- Llamadas a Métodos/Funciones

Inspeccionando estas construcciones, el visitor puede registrar las invocaciones estáticas que cada método o función registran dentro de su cuerpo. De esta manera puede generar las relaciones entre llamadores y llamados.

En cualquier tipo de lenguaje (principalmente aquellos con soporte multiparadigma), es posible encontrar archivos que contengan clases, métodos y funciones entremezclados (ver ejemplo en Figura 5.5). En este sentido, un aspecto fundamental para la correcta generación del GELMF es la determinación precisa de la signatura de cada artefacto de software analizado. De esta forma se garantiza la dependencia de cada método/función, es decir, a qué archivo y clase pertenece el artefacto. Es por esto que, si bien la información esencial está relacionada con las invocaciones, es importante destacar que el visitor debe recorrer numerosas construcciones del lenguaje, por ejemplo, las declaraciones de variables (para determinar los tipos de las mismas), las sentencias de importación de archivos (para conocer a qué archivo pertenecen los métodos/funciones referenciadas), las declaraciones de clases (para asociar a cada método a dicha clase), etc.

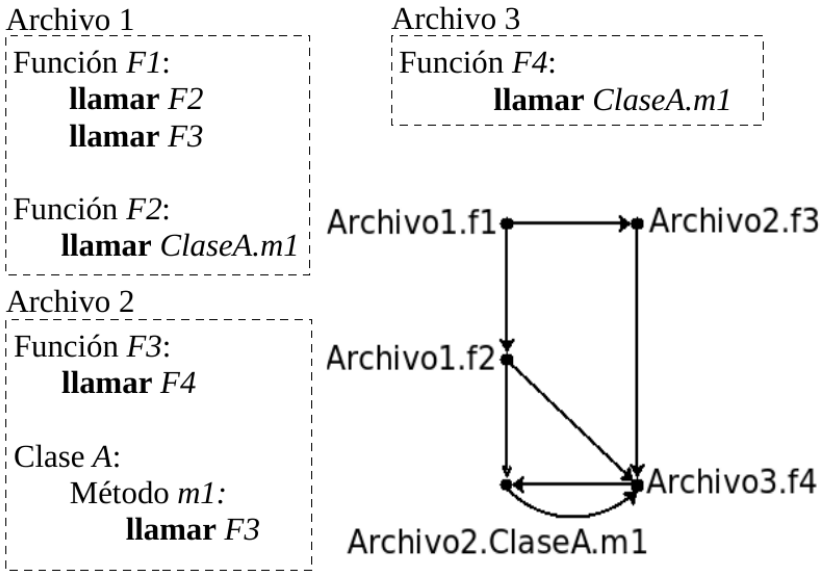

Figura 5.5: Ejemplo de construcción de GELMF a partir de pseudocódigo.

5.2.3. Árbol de Representación de la GUI

Las interfaces gráficas de usuario (o comúnmente referenciadas como *GUIs* por sus siglas en inglés), poseen una propiedad en común: todas poseen una estructura jerárquica. En términos generales, las distribuciones de los sistemas modernos poseen GUIs compuestas de ventanas que contienen diversos widgets tales como paneles, rótulos, barras de herramientas, menús y botones; algunos de estos pueden a su vez estar compuestos de otros widgets (por ejemplo, un botón puede integrar un rótulo y una imagen). En este sentido, más allá que ciertos componentes van variado entre las distintas plataformas, en general las formas de estructurarlas siempre han sido las mismas. Como ejemplo de lo antes mencionado, es posible asociar el mismo tipo de comportamiento a los botones en un sistema embebido, el hipertexto en un sistema web y las *actividades* en las plataformas móviles (como las aplicaciones en Android).

El ARGUI está basado en la estructura jerárquica estática natural de las interfaces gráficas contemporáneas. Este diseño ha demostrado ser lo suficientemente flexible como para trabajar con las GUI de hoy en día que continúan siendo jerárquicas a lo largo de los años. Este tipo de distribución se puede encontrar en las GUI basadas en web o GUI en plataformas móviles. Una estructura similar es utilizada por Menom y su grupo de

investigación en la herramienta *GUI Ripping Tool* (Memon y cols., 2003).

Cada nodo en el ARGUI representa un componente (o widget) de la GUI la cuales pueden ser clasificadas en:

- *contenedores,* como por ejemplo ventanas, paneles y diálogos; y
- *atómicos,* como por ejemplo botones, rótulos, imágenes y entradas de texto.

En este sentido, la estructura mantiene la representación estática de la GUI, reflejando la jerarquía implícita de los componentes. Para esto dicha estructura posee un único nodo raíz[4] que representa la aplicación y no posee ciclos (Memon y cols., 2003).

Por lo general, una vez que se invoca el sistema, se presenta al usuario una ventana de nivel superior (o un conjunto de ventanas). Todos los demás elementos de la GUI son utilizados por una de las ventanas de nivel superior o de sus descendientes. En este contexto, la jerarquía estática de la GUI podría extraerse en una estructura de árbol. La misma simplifica la navegación utilizando algoritmos de recorrido de árboles.

Formalmente, se define ARGUI como un árbol $T = (V, E)$, donde cada nodo en V representa un widget de la GUI y cada arco $E = (v, u)$, donde $v, u \in V$, representa que v contiene a u. Además, cada nodo contiene información referente al widget que representa tal como el tipo de componente, identificación, *tooltip*[5] y manejadores.

Naturalmente, algunos componentes pueden referenciar a otros (por ejemplo, un botón que al ser presionado abre una ventana), sin embargo la extracción de este tipo de información no es requerida para esta estructura ya que la misma ya se encuentra plasmada en el GELMF. Es por este motivo que la estructura extraída no contiene ciclos.

La jerarquía en el ARGUI es esencial a la hora de buscar términos relativos al Dominio del Problema con propósitos de rotulado. Frecuentemente, los componentes interactivos de la GUI proveen términos precisos que vinculan el comportamiento del mismo con una actividad del Dominio del Problema. Sin embargo, en ciertos casos el widget por sí mismo no contienen ningún rótulo o algún término sustancial para identificar la funcionalidad. En estos casos, los componentes de más bajo nivel (o incluso de más alto nivel) en la

[4]Un nodo que permite aglomerar todas las ventanas iniciales de la aplicación en una estructura.

[5]Un *tooltip* funciona al situar el cursor sobre algún widget, mostrando una ayuda adicional para informar al usuario de la finalidad del elemento sobre el que se encuentra.

jerarquía del ARGUI pueden proveer información más precisa acerca de la tarea que los componentes interactivos llevan acabo.

5.2.4. Construcción de ARGUI

La construcción del ARGUI puede llevarse a cabo usando alguna de estas dos estrategias:

- de la misma forma que el GELMF, se puede inspeccionar el AST en busca de aquellas funciones relacionadas con la creación y vinculación de elementos de la GUI, o
- para aquellos casos en que la GUI de aplicación haya sido desarrollada usando aplicaciones para construcción de GUIs[6] (Krause, 2007; Galde, 2016), se puede analizar el archivo generado por la aplicación con la que se construyó la GUI. Por lo general, este archivo es referenciado en el código fuente para cargar los widgets declarados y determinar las configuraciones y los comportamientos que los mismos tendrán.

Las dos opciones poseen ventajas y desventajas que la diferencian entre sí. Una de las ventajas de inspeccionar el AST es que dicha estructura provee más información que un archivo generado por los constructores de GUI con información exclusiva de la interfaz del sistema. Por lo general, los componentes GUI están más desagregados en el código fuente que en dicho archivo ya que en el mismo suelen plasmarse los componentes principales de la GUI, creando aquellos de menor importancia directamente en el código fuente.

Sin embargo, el análisis directo sobre el AST posee ciertas desventajas notables que dificultan la extracción de la jerarquía de la GUI, a continuación se mencionan algunas de las más relevantes:

- La mayoría de los lenguajes de propósito general no poseen construcciones específicas para crear componentes de la GUI. Por lo tanto, se requiere un análisis más

[6]Los constructores de GUI son herramientas de programación que simplifican la creación de interfaces gráficas de usuario, permitiéndole al diseñador ordenar los widgets con un editor del tipo WYSIWYG.

pormenorizado de los componentes del AST para detectar sentencias de creación o modificación de widgets.

- Construir la jerarquía de la GUI a partir del AST puede llegar a ser una tarea compleja debido a que dentro del mismo se pueden encontrar diversas construcciones que dificultarían la determinación del nivel de las componentes. Un claro ejemplo de esto son las clases anónimas, las cuales son creadas en cualquier contexto y pueden generar componentes GUI dentro de las mismas.

- Se pueden encontrar múltiples referencias a un mismo objeto de la GUI en diferentes archivos del código fuente.

Por otra parte, el análisis sobre el archivo del constructor de GUI se ve simplificado sobre el AST ya que toda la estructura principal de la GUI está plasmada en el mismo y cada componente pude ser fácilmente identificado en el proceso de análisis.

Con propósitos de simplificación del proceso de extracción de información, SAPSI obtiene información de la GUI analizando el archivo generado con el constructor de interfaces en conjunto con información encontrada en el código fuente.

En este tipo de archivos la jerarquía de la GUI es fácil de extraer ya que el diseño de las mismas siempre sigue un proceso *top-down*, es decir:

1. En primer lugar se definen los componentes contenedores de más alto nivel como por ejemplo las ventanas principales (*Frames*) o los diálogos (*Dialogs*). De acuerdo al tipo de sistema, se puede encontrar uno o varios. Por ejemplo, para un sistema en particular, este archivo puede tener como widget inicial la ventana principal de la aplicación y diferentes cuadros de diálogos que emergerán en diferentes circunstancias.

2. Posteriormente, se encuentran los contenedores intermedios, como los paneles, las barras de herramientas, etc. Estos están integrados en los contenedores de nivel superior.

3. Finalmente se pueden encontrar los componentes atómicos como botones, rótulos, campos de texto, listas despegables, etc.

Desde un punto de vista más técnico, este tipo de diseñadores GUI generan un archivo XML donde se almacenan los datos de la interfaz de usuario generada para el sistema. Como ejemplo de estas herramientas se pueden mencionar *JFormDesigner* para Java Swing, *QtDesigner* para Qt, *Glade* para GTK, entre otras. Este tipo de herramientas generan uno o varios archivos que contienen el diseño de la GUI, específicamente i) información de los widgets definidos y ii) cómo han sido desplegados los mismos. Posteriormente, el código fuente del sistema utiliza estos archivos para trabajar sobre la GUI diseñada.

En esta instancia, el front-end utiliza un parser XML, para analizar estos archivos y poder construir el ARGUI del sistema. Por medio de este analizador es posible inspeccionar todos los elementos de la GUI que vienen en el XML y construir el ARGUI plasmando toda la información contenida en dicho archivo. En el capítulo 8 se desarrolla más detalladamente los procesos ejecutados en la extracción de esta estructura desde el archivo generado. Es importante resaltar que este módulo del front-end depende de la librería gráfica y el lenguaje utilizados por el sistema y el mismo debe ser desarrollado y reemplazado de acuerdo a estos dos factores.

En la Figura 5.6 se puede observar una interfaz gráfica de una aplicación que implementa una libreta de contactos sencilla (dicho ejemplo está completamente desarrollado en el Anexo B). Es importante remarcar que cada uno de los nodos en el árbol contiene información relativa al widget que representa, como por ejemplo, nombre de la variable de identificación, tipo de widget (botón, entrada de texto, lista desplegable, ventana, etc.), rótulo (si es que el widget posee), manejador de eventos (por ejemplo, si es un botón, el nombre de la función o método que maneja el evento de botón presionado), etc.

5.2.5. Vinculación entre Representaciones

Una vez que se han creado las representaciones GELMF y ARGUI, es necesario integrar la información de manera tal de que los datos de dichas estructuras sean vinculados de alguna manera. Por ejemplo, si existe un método m en el código fuente que maneja un botón b de la GUI, entonces el nodo que representa al método m en el GELMF

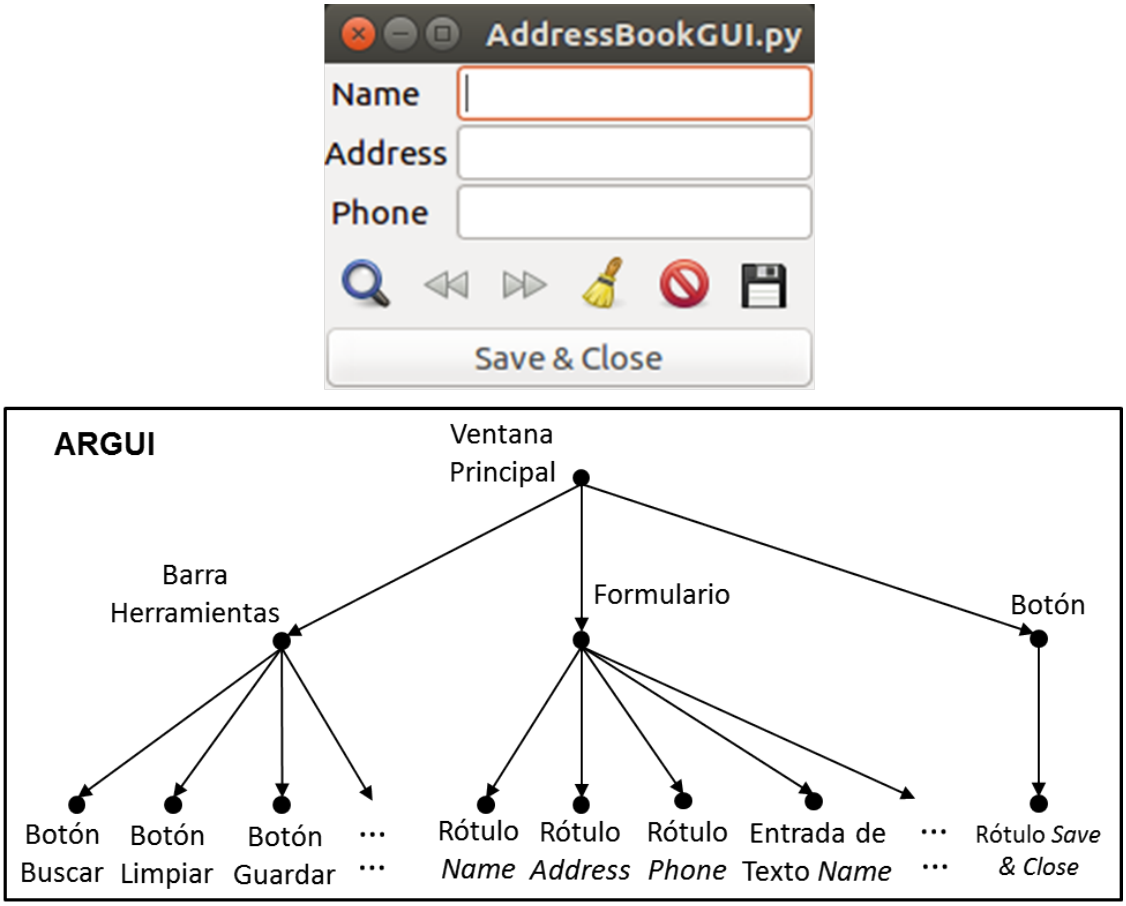

Figura 5.6: Ejemplo de ARGUI para GUI de la aplicación libreta de contactos.

debe estar vinculado con el nodo que representa al widget b en el ARGUI. Para esto, la estrategia lleva a cabo un conjunto de tareas elementales entre las que se encuentran:

- Detectar todos los widgets utilizados por cada método/función. Es decir, analiza el AST en busca de sentencias que "crean, leen o editan" artefactos de la GUI definidos en el ARGUI.
- Asociar cada componente GUI interactivo (como por ejemplo botones) del ARGUI con el método/función que implementa su comportamiento en el GELMF.

La vinculación entre los artefactos de las representaciones posibilita la navegación entre las mismas facilitando el proceso de análisis llevado a cabo en diferentes instancias durante el desarrollo de la estrategia. Por ejemplo, una vez encontrado el método/función que implementa el comportamiento de un botón determinado, con propósitos de análisis, es importante inspeccionar aquellos métodos/funciones que llaman o son llamados por el mismo. De la misma manera, si se está inspeccionado un nodo en el GELMF, es importante buscar aquellos widgets atómicos del ARGUI relacionados con dicho nodo en conjunto con los componentes contenedores que agrupan dichos widgets.

En el Anexo B se muestra un ejemplo completo de la aplicación de la estrategia a una aplicación sencilla que permite almacenar datos de contactos. Específicamente, la Sección A.1 exhibe el código fuente de la aplicación (desarrollada en lenguaje Python), las representaciones extraídas (GELMF y ARGUI) a partir del proyecto y el resultado de la vinculación entre ambas estructuras.

Otro de los principales motivos para integrar las estructuras es la extracción de un conjunto de métricas que posibilitarán caracterizar a cada nodo en el GELMF en relación a todo el sistema. La próxima sección describe las métricas utilizadas por SAPSI en sus diferentes etapas. Posteriormente, el Capítulo 6 explica detalladamente el método que permite agregar los valores de las métricas con el objetivo de calcular la métrica *Importancia Relativa*.

5.2.6. Extracción de Métricas de Software

Este apartado describe un conjunto de métricas de software utilizadas con diferentes propósitos durante las etapas de la estrategia de comprensión plateada. Es decir, cada

nodo del GELMF es atribuido con un conjunto de métricas de software las cuales caracterizan al nodo desde distintos puntos de vista. A su vez permiten estimar la IR de cada método/función con respeto a la lógica subyacente del sistema. Si bien el procedimiento para estimar IR se describe en el Capítulo 6, en este apartado se explica el conjunto de métricas de software de base extraídas para computar dicha métrica.

Peso GUI

Cada nodo en el GELMF contiene información acerca de los componentes con los cuales el método/función interactúa. Cada tipo de componente tiene un peso particular definido por la importancia que tiene ese componente desde la perspectiva del usuario. Es decir, para el usuario no es lo mismo un panel, cuyo propósito es agrupar otros componentes con algún fin, que un botón los cuales tienen asociados comportamientos y determinan las acciones del usuario para con el sistema. La Tabla 5.1 muestra diferentes tipos de componentes, algunos ejemplos de instancias específicas de cada grupo y el peso que SAPSI le asigna a cada grupo de widgets. Definiendo una taxonomía similar a la presentada por Cooper (Cooper y cols., 2014), se definió un esquema de pesos de acuerdo a cada tipo de componente. El esquema asigna pesos en el intervalo [1,3] adoptando como regla básica la siguiente premisa: mientras más significante sea el grupo de widgtes para el usuario, más alto debe ser el peso asignado para los mismos. Es por este motivo que aquellos widgets comprendidos en *Controles Imperativos* y *Controles Compuestos* poseen el mayor peso posible debido a que el usuario define el comportamiento del sistema a través de los mismos.

El argumento que subyace a esta métrica es el siguiente: tomando en cuenta que la interfaz gráfica es un componente estrechamente vinculado con el Dominio del Problema[7], es importante analizar los widgets que la integran y determinar aquellos componentes del Dominio del Programa que están relacionados en cierta forma a los mismos. En este sentido, los métodos/funciones que "interactúan" con los widgets de la GUI, son considerados importantes desde la perspectiva que plantea esta métrica ya que, de alguna manera, se están relacionando con información del Dominio del Problema.

[7]La GUI del sistema, contiene información y términos relacionados con el Dominio del Problema, de hecho, es por medio de la misma que el usuario puede resolver el problema para el cual el sistema fue desarrollado.

Tipo de Componente	Ejemplos de Instancias	Peso
Controles Imperativos	Botones / Hyperlinks / Items de Menú	3
Controles de Selección	*Check Boxes* / Listas / Listas Desplegables	2
Controles de Selección	*Check Boxes* / Listas / Listas Desplegables	2
Controles Compuestos	Selectores de Archivos / Tablas	3
Controles de Entrada	*Spinners* / Barras de Slide / Entradas de Texto	1
Controles de visualización	Barras de Scroll / *Splitters* / Rótulos	1
Menús	Lista de Menú / Menús Emergentes	1
Contenedores	Paneles / *Frames* / *Notebooks*	1
Errores, Alertas y Confirmaciones	Diálogos de Error / Diálogos de Información	2

Tabla 5.1: Pesos de los componentes de la GUI (*widgets*)

Código Fuente

Este tipo de criterios han sido considerados para proveer una estimación de la importancia de cada método/función con respecto al código fuente del sistema. Los mismos no requieren de explicaciones detalladas ya que son ampliamente conocidas en el contexto de Ingeniería de Software. Específicamente se consideran las siguientes:

- Complejidad Ciclomática de McCabe (CC)
- Número de Líneas de Código ((#LOCs))
- Visibilidad (V)
- Tipo (T)

Las dos primeras son ampliamente conocidas y brindan un estimativo de la complejidad estructural interna de cada método/función. Las otras dos brindan información de la accesibilidad de cada método/función en referencia a todo sistema. La *visibilidad*

registra si el método/función es público[8], mientras que el *tipo* es usado para definir si el nodo representa un método o una función.

Si bien la importancia de un método/función con respecto a todo el sistema no puede ser directamente asociada a este tipo de métricas, las mismas proveen buenos indicios acerca de la cantidad de "cómputo" que los mismos ejecutan. En este contexto, un método/función que obtiene altos valores para este conjunto de métricas, muy probablemente ejecute tareas más significativas que aquellos con valores bajos.

Grado de Utilidad Estructural (SUD)

Representa la métrica propuesta por Hamou-Lhadj y su grupo de investigación(Hamou-Lhadj y Lethbridge, 2006), que infiere cuando un nodo es estructuralmente más propenso a ser considerado como una utilidad, en relación a la "topología" del GELMF. El razonamiento elemental que los autores usan para determinar las utilidades es de la siguiente manera: *mientras más llamadas posea un método/función desde distintos lugares (mientras más arcos entren en el nodo), entonces muy posiblemente el nodo haya sido diseñado con múltiples propósitos, y por lo tanto, muy posiblemente dicho nodo deba ser considerado como una utilidad.*

$$SUD(r) = \frac{FanIn(r)}{\log_2(N)} * \frac{\log_2(\frac{N}{FanOut(r)+1})}{\log_2(N)} \tag{5.1}$$

En la Ecuación 5.1, r representa el nodo (método/función) para el cual el cálculo se lleva a cabo y N es el número total de nodos en el GELMF. $FanIn(r)$ indica el número de métodos/funciones que invocan el método/función analizado (grado de entrada del nodo) y $FanOut(r)$ indica el número de métodos/funciones que son invocados por el método/función (grado de salida del nodo).

De esta manera, si $SUD(r) = 0$, entonces r no es una utilidad; si $SUD(r)$ se aproxima a 1, entonces muy posiblemente lo sea[9]. La fórmula para el cálculo de $SUD(r)$ es explicada en detalle en (Hamou-Lhadj y Lethbridge, 2006).

[8]En el caso de las funciones, todas son consideradas públicas.

[9]Es importante resaltar que *SUD* sugiere la probabilidad de que un nodo sea una utilidad, mientras que *IR* representa la importancia relativa de un nodo con respecto a la lógica subyacente del sistema.

Intermediación (Betweenness Centrality)

Es una medida de centralidad que cuantifica la frecuencia o número de veces que un nodo actúa como "puente" en el camino más corto entre otros dos nodos.

$$BC(i) = \sum_{j \neq k} \frac{g_{jk}(i)}{g_{jk}} \tag{5.2}$$

en la Ecuación 5.2, $g_{jk}(i)$ es el número de caminos más cortos que conectan jk pasando a través de i, y g_{jk} es el número total de caminos más cortos. Cuando un nodo tiene un valor alto de BC es más probable que el mismo conecte con diversos grupos de nodos (Brandes, 2001). Usualmente, esta medida de centralidad debe ser normalizada por el número de pares de nodos excluyendo el nodo analizado:

$$BC'(i) = \frac{BC(i)}{[(n-1)(n-2)]/2} \tag{5.3}$$

SAPSI usa esta medida de centralidad (Ecuación 5.3) para reflejar cuando un nodo puede ser estructuralmente importante en relación a la topología general del GELMF.

Detalle de Implementación

Identifica cuando un nodo representa de manera directa un detalle de implementación, es decir, un método/función que ha sido declarado para facilitar el desarrollo del sistema, antes que resolver una tarea específica del Dominio del Problema. Como ejemplo de este tipo de funciones se pueden mencionar:

- métodos de acceso ($get < campo >$, $set < campo >$, $is < campo >$);
- constructores de clases;
- funciones integradas (*bult-in functions*);
- métodos/funciones, clases o archivos cuyos nombres contengan la palabra "utilidad", "útil", etc.

La mayoría de los lenguajes de programación proveen esta información en la signatura de los métodos/funciones, por lo tanto su detección es sencilla.

En el contexto de Ingeniería de Software, muchos trabajos emplean métricas de software para estimar el grado de mantenibilidad y complejidad de un sistema. Sin embargo, estos enfoques no ofrecen buenas correlaciones con el principal objetivo de SAPSI. El conjunto de métricas definido específicamente por la estrategia pretende inferir la influencia de cada método/función en referencia al "comportamiento" del sistema bajo análisis.

En el Capítulo 6 se detallan los aspectos relacionados con el propósito, la medición y comparación del conjunto de métricas antes descripto. Las técnicas y herramientas empleadas para la extracción de las métricas se detalla en el Capítulo 8. En el Anexo B se muestra un ejemplo de extracción de las métricas previamente descriptas para una aplicación sencilla que implementa una libreta de contactos.

5.3. Notas y Comentarios del capítulo

La extracción de información cumple un rol fundamental dentro de cualquier estrategia de CP. En este capítulo se describieron las TEI estática más utilizadas en el contexto de Ingeniería Reversa. Si bien se expusieron dos clases de técnicas, las más usadas para la implementación de estrategias de comprensión encontradas en el estado del arte están relacionadas con las técnicas de compilación. En pocas palabras, las mismas requieren de:

- un analizador lexicográfico (o lexer), que divide la cadena de entrada (código fuente) en tokens (unidades sintácticas atómicas);
- un analizador sintáctico (o parser), que analiza los tokens provisto por el lexer con el objetivo de instanciar los mismos en construcciones válidas del lenguaje de programación. Este, además construye un AST, es decir, una representación arbolada que abstrae la estructura del programa facilitando el análisis de la misma en busca de información del sistema.
- un módulo que permita recorrer la estructura generada previamente en busca de información de interés para la estrategia. Dicho módulo construye representacio-

nes que serán utilizadas como base para llevar a cabo el análisis dispuesto por la estrategia de comprensión.

Para el caso particular de SAPSI que posibilita analizar sistemas multiparadigma, usando esta técnica se construye un Grafo Estático de Llamadas a Método/Funciones. Además la estrategia también obtiene un árbol que representa la estructura jerárquica de la GUI del sistema. Esta representación contiene información de cada widget que compone la GUI en conjunto con la disposición de los mismos. Ambas representaciones son vinculadas entre sí para mejorar el acceso a la información en el sistema y proporcionar distintos aspectos del sistema bajo estudio, es decir, las mismas son utilizadas con diferentes propósitos a lo largo de todo el proceso de análisis planteado por SAPSI. Por otra parte, la integración de las mismas permite la extracción de un conjunto de métricas de software que servirán como base para el cálculo de una métrica de importancia para cada nodo del GELMF.

Todos los conceptos y proceso explicados en este capítulo posibilitan la extracción y representación de la información de base que SAPSI considera para llevar a cabo el análisis. Debido a que los procesos involucrados en la extracción de información están relacionados con aspectos de bajo nivel tales como la sintaxis del lenguaje utilizado o la librería gráfica usada para generar los componentes GUI, es importante separar las etapas centrales de análisis de la etapa de extracción. Es por este motivo que la estrategia plantea la siguiente separación de los procesos ejecutados por la estrategia:

- *font-end*: son los procesos que permiten obtener las representaciones de base que utiliza SAPSI: GELMF, ARGUI y métricas de software. Los conceptos y técnicas asociadas fueron descritos a lo largo de este capítulo. Estos procesos pueden ser intercambiados para lograr reconocer otros lenguajes y librerías gráficas.

- *back-end*: son los procesos que realizan el análisis sobre la información de base extraída por los procesos relativos al *front-end*. Los mismos serán descriptos en los capítulo subsiguientes.

Considerando este tipo de distribución, es posible anexar nuevos *front-ends* para que la estrategia sea aplicable a cualquier lenguaje con soporte multiparadigma. SAPSI es

implementada mediante un prototipo cuya arquitectura será exhibida en el capítulo 8; donde además se explicará con más detalle la división entre procesos *front-end* y *back-end*.

Capítulo 6

Reducción de Información

Menos del 10 % del código tienen que ver directamente con el propósito del sistema; el resto tiene que ver con entrada y salida, validación de datos, mantenimiento de estructuras de datos y otras labores domésticas.

— Mary Shaw

En un entorno industrial de desarrollo de software, las representaciones de base mediante las cuales cualquier estrategia de CP lleva a cabo su análisis podrían alcanzar un tamaño considerable. Por lo tanto, es importante que dichas estrategias incorporen métodos de reducción de información entre sus etapas (Hamou-Lhadj y Lethbridge, 2006; Spärck Jones, 2007). Dichos métodos deben reducir significativamente la pérdida de aquellos artefactos de software que sean sensibles a la lógica subyacente del programa. Para abordar este problema, se propone un método que brinda a la estrategia un criterio para discernir aquellos artefactos de software relevantes de aquellos que no lo son.

Este capítulo desarrolla el proceso de reducción de información adoptado por SAPSI. Durante esta fase, se realiza un análisis particular para eliminar métodos/funciones del GELMF, la cual es la principal representación sobre la cual se ejecuta el análisis propuesto por la estrategia. Aunque estos métodos/funciones están definidos en el código fuente del sistema, se ha demostrado que con propósitos de comprensión, los mismos pueden ser eliminados sin afectar el entendimiento general de la representación (Hamou-Lhadj

y Lethbridge, 2006; Spärck Jones, 2007; Koschke, 2002; Verwaest, 2007).

Para la detección de métodos/funciones, se utilizó un enfoque similar al propuesto por los autores Hamou-Lhadj y Lethbridge (Hamou-Lhadj y Lethbridge, 2006). En su trabajo, introducen un procedimiento que usa una métrica para resumir las trazas obtenidas de la ejecución de sistemas orientados a objeto, tomando como base el Grafo de Llamadas a Funciones del sistema. Para esto identifican artefactos de software que no son importantes para la lógica subyacente del programa; estos elementos son referenciados como *utilidades*. Sin embargo, la métrica propuesta por los autores no tiene en cuenta cierta información esencial relativa a cada artefacto de la representación de base, tal como el nombre de cada elemento, su visibilidad en referencia al sistema, métricas de software de código fuente, la relación con los componentes de la GUI, entre otros aspectos.

SAPSI introduce una métrica denominada IR, la cual determina para cada nodo r del GELMF la importancia del mismo en referencia a la lógica subyacente del sistema ($IR(r)$). Dicha métrica no sólo se utiliza como principal criterio de reducción de información, sino también para tomar decisiones durante el proceso de clustering (este se explica en el Capítulo 7). El valor $IR(r)$ representa la importancia de r dentro del sistema; por lo tanto, $1 - IR(r)$ representa el *grado de utilidad*[1] de r. En este sentido, si $IR(r)$ se acerca a 1, entonces r claramente no es una utilidad; si $IR(r)$ se acerca a 0, muy posiblemente lo sea. Para establecer qué métodos/funciones deben ser considerados como utilidades, es necesario definir un valor umbral identificado como $umbralUt$ (o *umbral de utilidad*). De este modo, si un nodo r obtiene un valor inferior para $IR(r)$ que $umbralUt$, entonces el método debe ser considerado como una utilidad.

En el contexto de Ingeniería de Software, la relevancia de un artefacto de software que integra un sistema en particular, debe ser determinada por diferentes aspectos que destaquen las distintos características del mismo. En este contexto, los tipos de métodos de evaluación más adecuados son aquellos que toman un conjunto de valores como entrada y devuelven un único valor como salida. Los métodos de decisión multicriterios se han utilizado en diversos trabajos que presentan estas características (Dujmović y cols.,

[1]El término *utilidad* es empleado con el mismo sentido que lo usan Hamou-Lhadj y Lethbridge en su trabajo (Hamou-Lhadj y Lethbridge, 2006). Por lo tanto, el *grado de utilidad* para r representa la probabilidad de que este nodo sea una *utilidad*.

s.f.; Miranda y cols., 2013; Berón y cols., 2016; Olsina y Rossi, 2002). En consecuencia, para calcular el valor de IR para cada nodo del GELMF, la estrategia utiliza un método de decisión multicriterio llamado LSP (Su y cols., 1987).

En lo que resta de este capítulo se explican los conceptos, procedimientos y estructuras asociadas al cálculo de la métrica IR usando LSP y el proceso de reducción de información llevado a cabo por SAPSI.

6.1. Un Método de Evaluación Multicriterio para Calcular la Importancia Relativa

Los Métodos de Evaluación Multi Criterio (MEMC) (también conocidos como Método de Decisión Multi Criterios) son utilizados para evaluación y toma de decisiones en el contexto de problemas que admiten un número finito de soluciones (Ballestero y Romero, 1998; Dujmović y Tré, 2011). Actualmente, existe una variedad de métodos utilizados en la toma de decisiones en diversas disciplinas (Triantaphyllou, 2000). Sin embargo, es difícil encontrar sistemas que implementaran estos métodos en la literatura disponible. Los métodos de este tipo más recientemente utilizados e implementados son ELECTRE (*ELimination Et Choix Traduisant la REalit*) y PROMETHEE (*Preference Ranking Organization METHod for Enrichment Evaluations*). Ambos métodos usan un enfoque similar al que plantea LSP. ELECTRE fue propuesto por Bernard Roy en 1971 (Roy, 1971). Las herramientas que implementan diferentes versiones de ELECTRE (Aguezzoul y cols., 2006; Montazer y cols., 2009; Mousseau y cols., 2000), presentan algunos inconvenientes, por ejemplo, utilizan una lógica difusa compleja con la que los usuarios deben lidiar y requieren una gran cantidad de interacción con el usuario. PROMETHEE fue desarrollado por Brans, y extendido por Vincke y Brans (Brans y cols., 1986). El método es bastante simple en concepción y aplicación cuando se compara con otros MEMC. Por este motivo es ampliamente utilizado en contextos de investigación y desarrollo. Dos de las implementaciones más utilizadas de PROMETHEE son *Decision LAB* y *PROCALC* (Brans y Mareschal, 2005). No obstante, ambos tienen inconvenientes similares a las implementaciones de ELECTRE. En el caso de implementaciones como AHP (Saaty,

1990, 2008) y MAUT (Keeney y Raiffa, 1976), no estaban disponibles para un análisis comparativo más profundo.

El método elegido para esta investigación es LSP, ya que existe una variedad de herramientas basadas en su implementación que pueden utilizarse con fines de evaluación, algunos ejemplos son: LSPmed (Dujmović y cols., s.f.), ISEE (Dujmović y Kadaster, 2002), NESSy (Miranda y cols., 2013), entre otros.

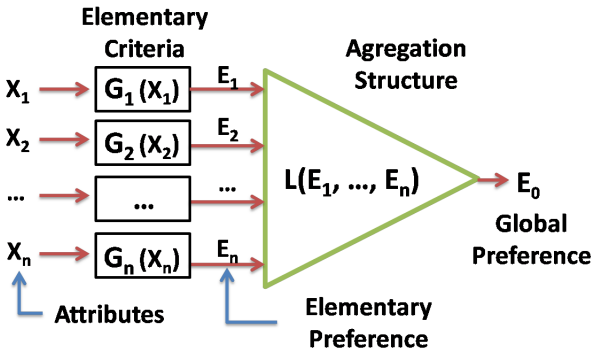

Figura 6.1: Representación del Método LSP

El método LSP está basado en un conjunto completo de atributos de entrada que reflejan con precisión todos los requisitos individuales planteados por el usuario. Una de las características más relevantes de este método es su capacidad para construir un modelo versátil y preciso de agregación de valores usando operadores lógicos.

La Figura 6.1 representa cómo funciona el método y cuáles son sus componentes esenciales. LSP usa un conjunto de elementos que se combinan sistemáticamente para obtener finalmente un valor que representa la importancia relativa de un nodo en particular en referencia al sistema en general. Estos elementos son:

i) un *Árbol de Criterios* (conjunto de requerimientos) que contiene los atributos mensurables de cada objeto a evaluar (ver Figura 6.1);

ii) un conjunto de *Criterios Elementales* (en la Figura 6.1 se muestran como $G_1, G_2..., G_n$), los cuales son funciones que normalizan el valor obtenido para cada atributos en el intervalo [0,1] (*Preferencias Elementales* en la Figura 6.1) y

iii) una *Estructura de Agregación* que agrega todas las Preferencias Elementales para obtener un valor final (*Preferencia Global* en la Figura 6.1).

La flexibilidad del modelo de evaluación planteado por LSP permite a los evaluadores

cuantificar con precisión el conocimiento que el experto disponible y de esta manera llevar a cabo una puesta a punto de requisitos complejos que reflejen con precisión las necesidades del usuario. Particularmente, este método fue considerado como la mejor alternativa para calcular IR en considerando las siguientes características principales:

- LSP permite la agregación de distintos valores, proporcionando al usuario una amplia gama de funciones lógicas.
- A diferencia de la mayoría de los métodos de evaluación, los cuales son efectivos para seleccionar la mejor opción de un conjunto de alternativas, LSP permite evaluar un único elemento por separado.

En las próximas secciones se explican todas las estructuras empleadas por el método para llevar a cabo el proceso de evaluación y finalmente obtener el valor de IR para cada nodo en el GELMF.

6.2. Árbol de Criterios

El primer paso en cualquier problema de decisión multicriterio es definir el conjunto de criterios con los que se debe evaluar el objeto bajo análisis. Este es un paso relativamente crítico, por lo que su ejecución no puede llevarse a cabo utilizando un procedimiento de modelado estándar, o un proceso sistemático. Esta tarea apela más al aspecto creativo de los métodos de decisión multicriterio que al sistemático. Este factor indica a los expertos en esta área que el paso más importante para resolver cualquiera de estos tipos de problemas es efectivamente, primero definir correctamente el problema (Triantaphyllou, 2000).

Con el objetivo de desarrollar un conjunto de criterios, se lleva a cabo un proceso de descomposición jerárquica para determinar claramente cuáles son los requerimientos que deben cumplir el objeto bajo análisis para ser calificado de acuerdo a lo estipulado por el experto evaluador (o en este caso, el ingeniero de software). En primer lugar, las características de alto nivel se definen y descomponen en subcaracterísticas. Al final de este proceso, se obtiene una lista de atributos mensurables donde cada uno tendrá asociada una *Variables de Performance* que registra el valor de la medición del atributo.

El resultado de esta tarea es un árbol que describe los requisitos principales que deben cumplir los objetos evaluados. El árbol de criterios de SAPSI se compone de las métricas detalladas en la Subsección 5.2.6. Esos atributos reflejan los requisitos que cada nodo debe cumplir para ser calificado como "importante" dentro del contexto del sistema.

6.3. Criterios Elementales

El método LSP requiere la normalización de los valores de las Variables de Performance correspondientes a los atributos en el Árbol de Criterios. Esta normalización es necesaria porque: i) en los contextos de evaluación se presentan diversas unidades de medición, y ii) los valores de las diferentes Variables de Performance pueden ser incomparables.

En LSP la normalización de los valores de los atributos considerados se logra mediante la definición de funciones de *Criterio Elemental*. Este tipo de funciones mapea un valor tomado por una Variable de Performance en otro valor contenido en el intervalo [0,1]. Este último representa el nivel de satisfacción de la variable de desempeño bajo observación. Este nivel de satisfacción se denomina Preferencia Elemental ($E_1, E_2, ..., E_n$ en la Figura 6.1). Este valor podría interpretarse como un porcentaje de los requisitos satisfechos. En este contexto, 0 representa una situación en la que la Variable de Performance no satisface los requisitos en absoluto, y 1 significa que el requisito está totalmente satisfecho (Su y cols., 1987). Cualquier función que mapee un valor (o un conjunto de valores) en el intervalo [0,1] se puede usar como función de criterios elementales. Muchos trabajos que usan LSP para realizar evaluaciones han utilizado diferentes tipos de funciones para lograr valores normalizados (Dujmović y cols., s.f.; Miranda y cols., 2013; Su y cols., 1987). Una de las más utilizadas es la que calcula una proporción basada en un valor alto en el contexto del atributo evaluado, es decir un valor que representa la completa satisfacción del atributo que se evalúa en relación a lo que se desea evaluar. Por ejemplo, si x denota la Variable de Performance para un atributo en particular, primero se puede determinar el valor máximo x_{max} que satisface completamente ese atributo, y el valor mínimo x_{min} que se considera demasiado bajo e inaceptable. La operación más simple que calcula el criterio elemental E como una función de x se puede definir como una

función creciente que consta de tres segmentos lineales, de la siguiente manera:

$$E = g_\lambda(x) = \begin{cases} 0, & \text{if } x \leq x_{min} \\ \frac{x}{x_{max}}, & \text{if } x_{min} < x \leq x_{max} \\ 1, & \text{if } x > x_{max} \end{cases}$$

Esta ecuación calcula una proporción de x, que representa la Variable de Performance, sobre el valor de x_{max}; y si x excede los valores extremos ($x \leq x_{min}$ o $x \geq x_{max}$), entonces el valor resultante $g_\lambda(x)$ se verá limitado con alguno de los valores extremos. Este tipo de función de criterio elemental es referenciada como "variable continua - multi-variable" (Su y cols., 1987) teniendo en cuenta la naturaleza de la Variable de Performance.

Algunos criterios específicos en realidad no requieren ninguna fórmula para obtener las preferencias, sino que toman dos valores posibles. Por ejemplo, si x representa la Variable de Performance para un atributo en particular, este tipo de función de Criterio Elemental asigna 1 en una situación particular y 0 en caso contrario. Este tipo de criterios elementales se puede definir usando la siguiente función:

$$E = f_\lambda(x) = \begin{cases} 0, & \text{si } x \models \beta \\ 1, & \text{si } x \models \alpha \end{cases}$$

donde α y β son proposiciones lógicas que representan situaciones donde x puede o no satisfacer las mismas. Este tipo de función de Criterio Elemental es referenciada como "discreta - multinivel" (Su y cols., 1987). En este caso en particular, algunos trabajos se refieren a la misma como "discreta - binaria" ya que sólo puede tomar uno de dos valores posibles.

SAPSI define una función de Criterio Elemental para cada atributo definido en el Árbol de Criterios (métricas definidas en la Subsección 5.2.6). A continuación se especifican las funciones para todas las Variables de Performance en el Árbol de Criterios:

Métricas de Código Fuente - Peso GUI: El Criterio Elemental es de tipo "continuo variable - multi-variable". Empleando la función $g_\lambda(x)$ donde $\lambda = p_{gui}$, x denota el peso GUIpara un método/función particular. Como se explicó en la Sección 5.2.6, el valor de x es la suma de los puntajes para cada widget que el méto-

do/función utilizó con cierto propósito. En este sentido, se definen los siguientes valores extremos teniendo en cuenta el siguiente razonamiento:

- x_{min} representa cuando un método/función no utiliza ningún tipo de componente GUI. En este caso x_{min}=0.
- x_{max} modela cuando un método/función tiene cierta interacción con widgets de la GUI del sistema, es decir que usa o modifica de alguna manera la interfaz de usuario. A estos efectos, este valor debe reflejar cuándo el método/función bajo evaluación "utiliza" componentes de la GUI y de esta forma ser considerado un artefacto relevante para el sistema desde el punto de vista de esta métrica. La suma de pesos se calcula teniendo en cuenta el esquema de ponderación definido en 5.2.6. Teniendo en cuenta los aspectos antes mencionados, un valor extremadamente bajo podría derivar en que métodos/funciones que usan muy pocos widgets (o componentes irrelevantes) sean considerados como relevantes para el sistema, mientras que un valor alto aumenta las expectativas para cada método/función y sólo aquellos que tengan una alta interacción con la sean considerados relevantes.

  En este contexto se asigna $x_{max} = 5$ ya que es un valor que representa cuando el método/función ha interactuado con un conjunto de widgets con un peso que debe ser tenido en cuenta. Es importante destacar que en las aplicaciones de gran tamaño, la cantidad de métodos/funciones que interactúan con la GUI (al menos con un Peso GUI de 1) es considerablemente menor en relación a la cantidad total del sistema. Es por este motivo que se plantea un valor que posibilite medir una interacción "básica" del nodo con los elementos de la GUI, de acuerdo al esquema de ponderación definido en 5.2.6.

Métricas de Código - CC: El Criterio Elemental es del tipo "continuo - multivariable". Haciendo uso de la función $g_\lambda(x)$, donde $\lambda = CC$, x es la Complejidad Ciclomática de McCabe medida para un método/función particular y se define los valores extremos de la siguiente manera:

- x_{min} es el valor más bajo de CC que un método/función puede obtener. En

este caso x_{min}=1.

- x_{max} refleja el valor que debe ser considerado como alto para CC para todos los sistemas bajo análisis. En este entorno, si bien algunos trabajos han definido rangos que reflejan el grado de mantenibilidad de una función, los mismos no son aplicables para el cálculo de IR y los objetivos de investigación planteados en este trabajo. Los diversos sistemas presentan métodos/funciones con esquemas de complejidad disímiles. Es por esto que la evaluación del valor de CC para cada método/función, depende de la complejidad general que presenta cada sistema (Mathias y cols., 1999; Erdogmus y Tanir, 2002). En este contexto, se estima a x_{max} como el valor promedio de Complejidad Ciclomática para todo el sistema bajo estudio. Para la estimación de este promedio, se toma en consideración los métodos/funciones cuyo valor de CC es mayor que 1, de lo contrario el promedio podría verse seriamente afectado por esos métodos/funciones irrelevantes, por ejemplo, métodos de acceso (como `getXX`, `setXX` o `isXX`), inicializadores (como los constructores de aquellas clases del modelo entidad-relación) y funciones incorporadas del lenguaje.

Métricas de Código - #LOCs: El Criterio Elemental es del tipo "continuo - multivariable". Si se emplea la función $g_\lambda(x)$ donde $\lambda = \#LOCs$, x es el número de líneas de código (sin considerar comentarios) para un método/función particular, y se establecen los siguientes valores extremos:

- x_{min} es la menor cantidad de líneas de código que puede tener un método/función. Para este caso x_{min}=2[2].
- x_{max} refleja el valor que debe considerarse como una gran cantidad de líneas de código para un método/función particular, es decir un valor alto de $\#LOCs$, teniendo en cuenta el tipo de sistema bajo análisis. De la misma manera que con CC, los diferentes sistemas pueden presentar tamaños de métodos/funciones disímiles en LOCs. Es por esto que debe considerarse que la

[2]Es el menor valor que puede alcanzar la Variable de Performance para $LOCs$ ya que se cuenta la línea de la signatura. También se consideran los casos de ciertos lenguajes como Python que el mismo no permite que el cuerpo del método esté vacío, en ese caso x_{min}=2

evaluación de cada método/función con respecto a $\#LOCs$ depende del tamaño general, considerando líneas de código, del sistema en su totalidad. La métrica $\#LOCs$ presenta una situación similar a CC, por lo tanto una forma más precisa de estimar x_{max} es considerar el número promedio de líneas de código para todo el sistema bajo análisis. Al igual que CC, también se consideran los métodos/funciones cuyos $\#LOCs$ son mayores que x_{min} para la estimación promedio.

Métricas de Código - Visibilidad: El tipo de Criterio Elemental es "discreto - binaria". Si se emplea la función $f_\lambda(x)$ donde $\lambda = visibilidad$, si x representa la visibilidad de un método/función particular, entonces $f_{visibilidad}(x)$ asigna 0 cuando el nodo representa un método privado o protegido (β). La función asigna 1 en cualquier otro caso (α). El razonamiento subyacente de esta interpretación es que aquellos métodos/funciones que pueden ser llamados desde fuera del alcance definido por el archivo o la clase correspondiente, son más relevantes que el resto considerando un criterio de accesibilidad.

Métricas de Código - Tipo: El tipo de Criterio Elemental es "discreta - binaria". Si se usa la función $f_\lambda(x)$ donde, $\lambda = tipo$, si x representa el tipo de un método/función particular, entonces $f_{tipo}(x)$ asigna 1 cuando el nodo representa un método (β). La función asigna 0 en cualquier otro caso (α). Para este criterio se aplicó un razonamiento similar que *visibilidad*, es decir, a través del mismo se pondera que el método/función sea un método antes que una función. El razonamiento detrás de este criterio está relacionado con el paradigma orientado a objetos, ya que en los mismos la principal "unidad funcional" es el método. En lenguajes con soporte multiparadigma como Python, es posible definir funciones en sistemas desarrollados mayormente siguiendo el paradigma orientado a objetos, aunque las mismas carecen de la relevancia y la semántica de los métodos. Por lo general, estas son definidas para llevar a cabo acciones complementarias o usadas frecuentemente por distintos métodos/funciones (este funcionamiento es asociado por varios autores a las *utilidades* (Hamou-Lhadj y Lethbridge, 2006)). Es decir, desde la perspectiva que intenta medir esta métrica, en un sistema orientado a objeto, las funciones

cumplen un rol secundario en contraste con los métodos que conforman las clases, entidades habitualmente relacionadas al Dominio del Problema. Por otra parte, en el caso en que el sistema esté desarrollado exclusivamente usando funciones, este criterio no afecta la evaluación de cada entidad.

Grado de Utilidad Estructural (SUD): El Criterio Elemental es del tipo "continuo variable - directo". Como lo detallan sus autores, el cálculo de esta medida retorna valores normalizados en el intervalo [0,1] (Hamou-Lhadj y cols., 2005). Dado que la interpretación de la métrica IR es la importancia relativa de un artefacto de software en referencia a la lógica subyacente del sistema, es necesario considerar la inversa del valor de utilidad estructural. De esta manera, la Preferencia Elemental resultante representa a un método/función estructuralmente relevante si dicho valor se acerca a 1, mientras que si se acerca a 0 representa todo lo contrario. Por lo tanto, esta función de Criterio Elemental invierte el valor normalizado, es decir, teniendo en cuenta el intervalo [0,1], si $SUD(r)$ es el valor de utilidad estructural para el nodo r, la Preferencia Elemental para esta variable de performance será $1 - SUD(r)$.

Intermediación (Betweenness Centrality): El Criterio Elemental es del tipo "continuo variable - directo". El cálculo de esta reconocida medida de centralidad retorna valores normalizados en el intervalo [0,1] (Brandes, 2001).

Detalle de Implementación: El tipo de Criterio Elemental es "discreto - binaria". Si se emplea la función $f_\lambda(x)$ donde $\lambda = detalle$, si x representa si el método/función es un detalle de implementación, entonces $f_{detalle}(x)$ asigna 0 cuando el nodo representa un detalle de implementación (β), según lo especificado en la definición de esta métrica. La función asigna 1 en cualquier otro caso (α). Más adelante en la sección se puede comprobar que este criterio será fundamental a la hora del filtrado de aquellos nodos que representen detalles de implementación.

Como se puede observar, algunas funciones de Criterio Elemental usan un máximo (x_{max}) que determina un valor que satisface completamente el atributo medido. Además, este valor es utilizado para determinar la proporción de x con respecto a x_{max}. En el

contexto de medición y evaluación en Ingeniería Reversa, valores máximos y mínimos han sido frecuentemente utilizados para identificar el rango de valores considerados como aceptables o inaceptables para un atributo determinado (Erdogmus y Tanir, 2002; Shatnawi, 2015). En este trabajo doctoral, tales valores han sido definidos tomando como base la experiencia del equipo de investigación, evaluaciones experimentales con la estrategia e interacción con expertos en evaluación de sistemas de software.

Las funciones de Criterio Elemental presentadas en este trabajo son instancias de una amplia gama de tipos de funciones que pueden ser utilizadas con estos propósitos. El lector interesado en profundizar conocimiento sobre dichas funciones puede leer los trabajos de Dujmovic (Su y cols., 1987; J. j. Dujmovic y Elnicki, 1982) Ballestero (Ballestero y Romero, 1998) y Miranda (Miranda y cols., 2013).

6.4. Estructura de Agregación

La Estructura de Agregación es el componente esencial en el método LSP ya que constituye el modelo de evaluación real. Las Preferencias Elementales que resultan de la aplicación de los Criterios Elementales a los atributos mensurables se deben agregar para obtener una Preferencia Global única (ver la Figura 6.1). Este es el grado de satisfacción del nodo bajo evaluación con respecto a todos los requerimientos plasmados en el Árbol de Criterios. En el contexto de SAPSI, la Preferencia Global representa el valor de *Importancia Relativa (IR)* para ese nodo.

Para alcanzar la Preferencia Global, se utilizan algunas funciones de agregación (operadores). Estas funciones reciben un conjunto de Preferencias Elementales y sus correspondientes ponderaciones como entrada. Los pesos representan la importancia relativa para cada preferencia, es decir cual es la importancia que posee cada atributo con respecto a los demás. Las funciones devuelven preferencias agregadas como sus salidas, por lo tanto estos valores se encuentran en el intervalo [0,1]. Posteriormente, todas las salidas se agregan en el siguiente nivel de la estructura. Este proceso se repite hasta que se alcanza la Preferencia Global, es decir el resultado de computar aquel operador que condensa todos los demás, en otras palabras, el último operador de la estructura. Cada operador

en una Estructura de Agregación de LSP se puede expresar con la Ecuación 6.1.

$$E = (w_1 e_1^r + w_2 e_2^r + ... + w_k e_k^r)^{\frac{1}{r}} \tag{6.1}$$

donde:

$$-\infty \leq r \leq +\infty$$
$$0 \leq w_i \leq 1 \text{ y } i = 1..k$$
$$w_1 + ... + w_k = 1$$

La elección de r determina la ubicación de E entre el valor mínimo $E_{min} = min(e_1, ..., e_n)$ y el valor máximo $E_{max} = max(e_1, ..., e_n)$. Para $r = -\infty$, la media pesada se reduce a la conjunción pura (la función *mínimo*), y para $r = +\infty$ la media pesada se reduce a la disyunción pura (la función *máximo*). Por lo tanto, E puede concebirse como un esquema general de instanciación que produce un espectro continuo de funciones de agregación, según el valor de r (J. j. Dujmovic y Elnicki, 1982).

El rango entre la conjunción pura y la disyunción pura suele estar cubierto por una secuencia de operadores ubicados equidistantemente denominados: C, $C{+}{+}$, $C{+}$, $C{+}{-}$, CA, $C-+$, $C-$, $C--$, A, $D--$, $D-$, $D-+$, DA, $D+-$, $D+$, $D++$ y D. Algunos operadores modelan requisitos obligatorios, esto significa que si alguno de los e_n valores de entrada es 0, entonces la preferencia de salida resultante E es 0. Los valores de r permiten generar muchas operaciones conocidas como funciones de Conjunción/Disyunción Generalizadas (CDG). La fórmula empleada para calcular los valores en la Tabla 6.1 se describe ampliamente en (J. j. Dujmovic y Elnicki, 1982; J. J. Dujmovic, 2008). La Tabla 6.1 muestra algunos de los valores más relevantes de r, teniendo en cuenta la cantidad de valores de entrada n. La parte sombreada muestra aquellos operadores que son obligatorios. Dujmovic y su grupo de investigación (J. j. Dujmovic y Elnicki, 1982; J. J. Dujmovic, 2007) presentan una tabla extendida con más operadores y valores; la misma se puede consulta en el Anexo B.

Para mostrar un ejemplo específico del cálculo de un operador, la Ecuación 6.2 calcula la preferencia parcial para el operador $D-$ que recibe los siguientes cuatro valores y pesos como entrada (como se representa en la Figura 6.2): i) preferencia elemental: 7.9, peso: 43/100; ii) preferencia elemental: 14.8, peso: 31/100; iii) preferencia elemental: 100, peso:

13/100; y iv) preferencia elemental: 100, peso: 13/100.

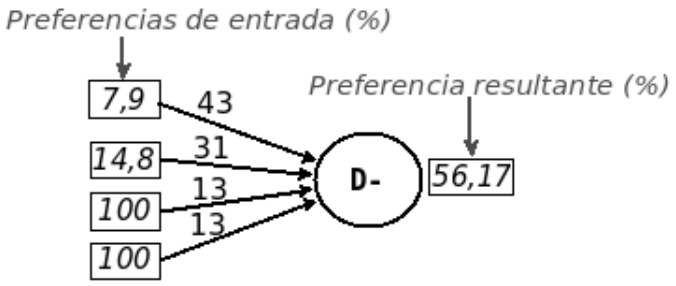

Figura 6.2: Ejemplo de operador LSP ($D-$)

$$(0{,}43 * (7{,}9^{2{,}3}) + 0{,}31 * (14{,}8^{2{,}3}) + (0{,}13 * (100^{2{,}3}) + 0{,}13 * (100^{2{,}3})^{\frac{1}{2{,}3}} = 56{,}17 \quad (6.2)$$

Los operadores conjuntivos, también conocidos como operadores de simultaneidad, se emplean cuando se desea una satisfacción simultánea de todos los requerimientos en un grupo. Los operadores disyuntivos, también referidos como operadores de reemplazo, se utilizan siempre que una alta satisfacción de cualquier requerimientos pueda reemplazar parcial o completamente la satisfacción de todos los demás requerimientos del grupo. El operador de neutralidad, o media aritmética, se encuentra entre la capacidad de sustitución y la simultaneidad. También es posible usar algunas funciones especiales agregar preferencias, como la función cuadrática (SQU), armónica (HAR) o geométrica (GEO). El comportamiento de estos operadores es explicado con más detalle por Dujmovic en (Su y cols., 1987).

Teniendo en cuenta el grupo de atributos que se deseen agregar, se deben definir dos parámetros que modelan el comportamiento de cada operador: i) un valor que representa la importancia relativa de cada entrada, también referenciados como *pesos*, y ii) el grado de polarización del operador, en otras palabras, el grado de conjunción/disyunción o dicho de otra forma, grado de "simultaneidad/reemplazabilidad" (ver Anexo B).

Operador	Símbolo	Grado Dis.	r		
			n=2	n=3	n=4
Disyunción	D	1.0000	$+\infty$	$+\infty$	$+\infty$
QD Fuerte(+)	$D++$	0.9375	20.63	24.3	27.71
QD Fuerte	$D+$	0.8750	9.52	11.09	12.27
QD Débil(+)	$D-+$	0.6875	2.79	3.1	3.32
QD	$D-$	0.6250	2.02	2.19	2.3
Media Aritmética	A	0.5000	1.00	1.00	1.00
QC Débil(-)	$C--$	0.4375	0.62	0.57	0.55
QC Débil	$C-$	0.3750	0.26	0.19	0.15
QC Débil(+)	$C-+$	0.3125	-0.15	-0.21	-0.24
QC Fuerte	$C+$	0.1250	-3.51	-3.11	-2.82
QC Fuerte(+)	$C++$	0.0625	-9.06	-7.64	-6.69
Conjunción	C	0.0000	$-\infty$	$-\infty$	$-\infty$

Tabla 6.1: Valores para r correspondientes a cada función CDG y el número de operandos (n). La abreviación "QD" significa Cuasi Disyunción mientras que "QC" representa Cuasi Conjunción. La columna 'Grado Dis.' muestra el grado de "simultaneidad" del operador, es decir el grado de disyunción.

6.4.1. Definición de Pesos de las Entradas

Un paso fundamental al emplear el método LSP (y también para cualquier tipo de método multicriterio) es definir de manera precisa los datos pertinentes. Frecuentemente en los problemas en el marco de Métodos de Evaluación Multi-Criterios los datos no pueden ser conocidos en términos de valores absolutos, es decir, es necesario tener en cuenta muchos factores relacionados con el contexto de evaluación. Por ejemplo, es necesario saber cuál es la importancia relativa del tipo específico de elemento bajo análisis con respecto a un atributo determinado. Aunque ciertos aspectos como el anterior pueden ser esenciales para alcanzar el objetivo que persigue la evaluación, es bastante complejo, si no imposible, cuantificarlos adecuadamente. Por lo tanto, muchos métodos de toma de decisiones intentan determinar la importancia relativa, o el peso, de las alternativas en

términos de cada criterio involucrado en un problema de evaluación con múltiples criterios. El enfoque basado en las comparaciones por pares propuestas por Saaty (Saaty, 1990, 2008) como parte del método AHP, atrajo el interés de muchos investigadores y profesionales debido a su fácil aplicabilidad y sus propiedades matemáticas. Las comparaciones por pares se pueden usar para determinar la importancia relativa de cada atributo en términos de una característica más alta. Empleando este enfoque, el evaluador debe expresar lo requerido respecto del valor de una comparación por pares a la vez.

La determinación de los pesos en la Estructura de Agregación de para calcular IR se llevó cabo usando el enfoque del "valor propio" o *eigenvalue*, propuesto primeramente en el marco del método AHP (Saaty, 1990, 2008; Triantaphyllou, 2000). Este enfoque genera una matriz $n*n$ $\boldsymbol{A}$ de comparaciones por pares, donde n es el número de criterios de evaluación considerados. Cada entrada a_{jk} de la matriz $\boldsymbol{A}$ representa la importancia del j-ésimo criterio con respecto al k-ésimo criterio. Si $a_{jk} > 1$, entonces el j-ésimo criterio es más importante que el k-ésimo, mientras que si $a_{jk} < 1$, entonces el j-ésimo criterio es menos importante que el k-ésimo. Si dos criterios poseen la misma importancia, entonces la entrada para a_{jk} es 1. Las entradas a_{jk} y a_{kj} satisfacen las siguiente restricción $a_{jk} * a_{kj} = 1$ [3].

La importancia relativa entre dos criterios es medida de acuerdo a una escala numérica de 1 a 9 como se muestra en la Tabla 6.2, donde se asume que el j-ésimo criterio es igual o más importante que el k-ésimo (Saaty, 1990, 2008). El contenido de la columna "Definición" describe de manera tentativa la evaluación cualitativa del experto en relación a la importancia relativa entre dos criterios. Esta información define brevemente el valor numérico de la columna "Intensidad". También es posible asignar valores intermedio los cuales no se corresponden con una interpretación específica.

Una vez que se construye la matriz $\boldsymbol{A}$, es posible derivar la matriz de comparación por pares normalizada $\boldsymbol{A}_{norm}$ haciendo que la suma de las entradas en cada columna sea igual a 1. Esto significa que cada entrada a_{jk} de la matriz $\boldsymbol{A}_{norma}$ se calcula de la siguiente manera:

[3]Claramente, a_{jj}=1 para todos los j.

Intensidad de la Importancia	Definición	Explicación
1	Importancia equilibrada	Dos actividad contribuyen de manera equitativa al objetivo
3	Importancia moderada	La experiencia y el juicio favorecen ligeramente una actividad sobre otra
5	Importancia marcada	La experiencia y el juicio favorecen fuertemente una actividad sobre otra
7	Importancia demostrada	Una actividad es favorecida fuertemente por sobre la otra; su dominio es demostrado en la práctica
9	Importancia extrema	La evidencia que justifica cómo una actividad se ve favorecida por sobre la otra es del orden de afirmación más alto posible
Recíprocos de valores anteriores	Si la actividad i tiene uno de los números anteriores distintos de cero asignados cuando se compara con la actividad j, entonces j tiene el valor recíproco cuando se compara con i	Una suposición evidente

Tabla 6.2: Resumen de esquema de pesos relativos

$$\overline{a}_{jk} = \frac{a_{jk}}{\sum_{l=1}^{n} a_{lk}}$$

Finalmente, el vector de pesos de los criterios $\boldsymbol{w}$ (que es un vector n-dimensional) se construye realizando un promedio de las entradas en cada columna de $\boldsymbol{A}_{norm}$,

$$\overline{w}_j = \frac{\sum_{l=1}^{n} \overline{a}_{jk}}{n}$$

Este enfoque también proporciona una técnica efectiva para verificar la consistencia de las evaluaciones hechas por el experto evaluador al construir cada una de las matrices

de comparación por pares involucradas en el proceso. La explicación de dicha técnica está fuera del alcance de este trabajo y puede ser consultada con más profundidad en los trabajos referidos al método AHP (Saaty, 1990, 2008; Triantaphyllou, 2000).

Aunque el enfoque utilizado para derivar los pesos está influenciado por las decisiones del evaluador, el mismo ha sido ampliamente utilizado ya que reduce en gran medida los efectos de los errores en la estimación de pesos.

6.4.2. Estimación del Nivel de Polarización para cada Operador

Como casi todos los MEMC, LSP requiere de una serie de decisiones por parte de los expertos evaluadores. Específicamente, la definición de la Estructura de Agregación requiere la especificación de un conjunto de operadores lógicos. En este sentido, una de las características interesantes del método LSP en contraste con los métodos meramente lineales y de *scoring*, reside en la capacidad de tratar con diferentes relaciones lógicas para modelar los requisitos de los expertos, a saber:

- *Simultaneidad:* cuando se detecta que dos o más preferencias de entrada deben estar presentes de manera simultánea.
- *Reemplazabilidad:* cuando se distingue que se pueden alternar dos o más preferencias de entrada.
- *Neutralidad:* cuando se interpreta que dos o más preferencias de entrada se pueden agrupar independientemente (ni relaciones conjuntivas ni disyuntivas).
- *Relaciones simétricas:* cuando se percibe que dos o más preferencias de entrada afectan la evaluación de igual manera lógica (aunque puede que con diferentes pesos).
- *Relaciones asimétricas:* cuando los atributos obligatorios se combinan con atributos deseables u opcionales y viceversa.

Es posible definir una combinación de funciones anidadas para modelar relaciones de mayor complejidad (como con las relaciones asimétricas (J. J. Dujmovic, 2008; Su y cols., 1987)). Debido a las características de los criterios considerados y el comportamiento

requerido para el grupo de operadores necesarios que computarán los valores de IR, la Estructura de Agregación definida por SAPSI para el calcula de dicha métrica no incorpora funciones simétricas ni asimétricas.

Al igual que la definición de peso de entrada, la selección del operador de LSP también implica el juicio del experto evaluador. Las pautas para seleccionar operadores en la Tabla 6.1 incorporan las siguientes decisiones básicas:

- Decidir si la polarización de un bloque de agregación es conjuntivo (para modelar la simultaneidad de los atributos), disyuntivo (para modelar la capacidad de reemplazo de los atributos) o neutral. La selección entre polarización conjuntiva y disyuntiva regularmente se realiza sin errores (Dujmović, 2013).

- Identificar si existen preferencias de entrada obligatorias.

- Seleccionar el tipo de función con respecto a la relación entre los atributos de entrada. Es decir, evaluar si la relación se modela con un operador simple, simétrico o asimétrico.

- Seleccionar el operador según el grado de intensidad de la polarización lógica (*strong*, *medium* o *weak*).

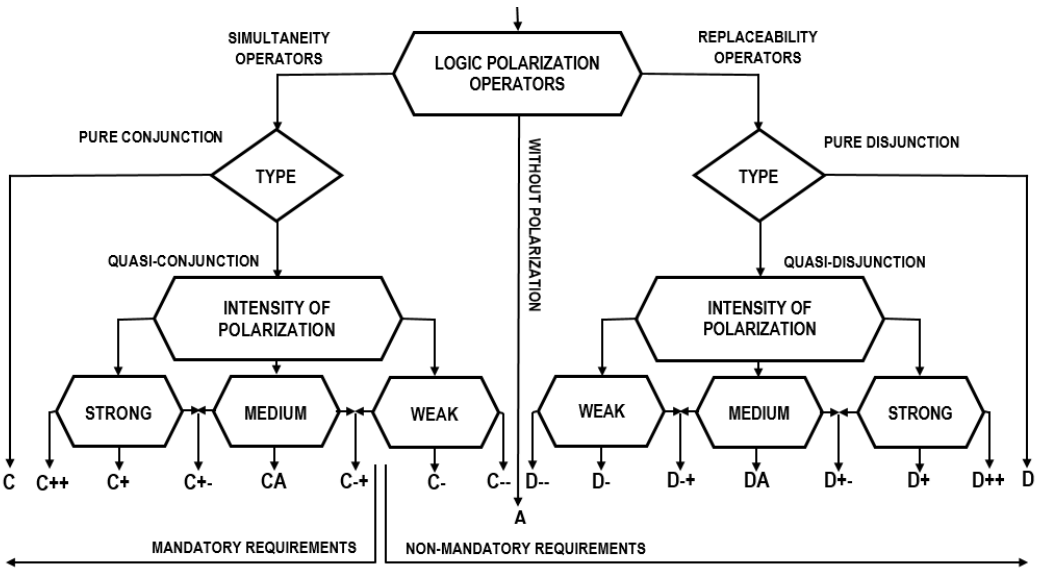

Figura 6.3: Operadores de LSP conjuntivos y disyuntivos y niveles de polarización.

La Figura 6.3 muestra un modelo de 17 niveles de operadores que representan funciones generalizadas conjuntivas y disyuntivas (aunque se puede usar un modelo de 25 niveles si se necesitara más precisión, ver Anexo B). Esta figura representa las decisiones básicas que deben analizarse al seleccionar cada operador de LSP. Por ejemplo, el primer paso para modelar la simultaneidad es decidir si usar operadores del tipo de $C--$ y $C-$ que modelan una polarización conjuntiva débil sin requisitos obligatorios, o operadores como $C-+$ que modelan una polarización conjuntiva fuerte con atributos obligatorios (Dujmović, 2013). Es evidente que estos son lineamientos y la determinación final del operador involucra la decisión del evaluador. En consecuencia, se supone que dicho evaluador es un experto en el contexto de evaluación del objeto bajo estudio y sus interpretaciones deben ser adscritas a la forma en que se agregan las preferencias (Dujmović, 2013).

En el siguiente apartado se presenta la Estructura de Agregación que modela la métrica Importancia Relativa además de algunos ejemplos con respecto a las técnicas y el razonamiento aplicado por los expertos evaluadores en este caso.

6.4.3. Estructura de Agregación para Computar la Importancia Relativa

SAPSI define métricas que intentan identificar la importancia de cada nodo dentro del sistema. Por lo tanto, en este caso particular, la Estructura de Agregación está compuesta por operadores que propagan todos los valores de satisfacción con el objetivo de obtener una Preferencia Global que refleje la IR para cada nodo en relación a todo el sistema. La Figura 6.4 muestra la Estructura de Agregación definida para calcular dicha métrica. Los párrafos subsiguientes explican el razonamiento detrás de la elección de los operadores y los pesos de esta estructura.

Las métricas del código fuente son agregadas por el operador $D-$ (Cuasi Disyunción débil), el cual es un operador disyuntivo débil similar a la Media Aritmética (A), excepto que el primero modela cierta simultaneidad en las entradas. El razonamiento detrás de la selección de este operador es que este conjunto particular de métricas no presenta una situación en la que se deba aplicar una polarización *fuerte* o *media*. Por lo tanto, se

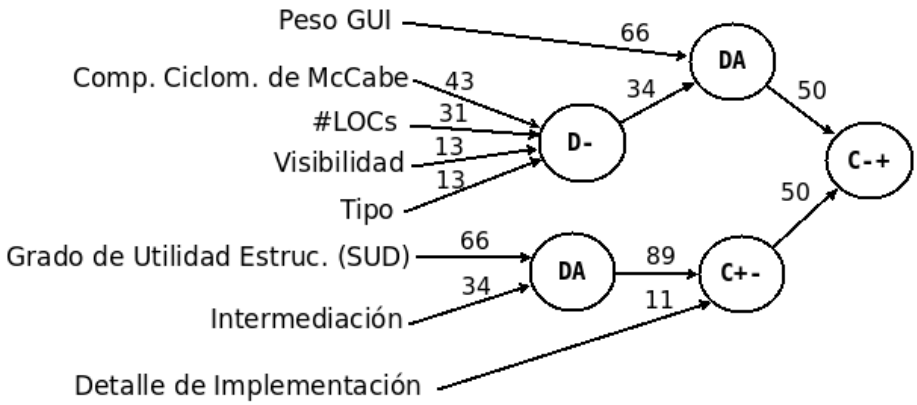

Figura 6.4: Estructura de Agregación para computar IR

modela esta situación con el operador $D-$ que, dentro del conjunto de funciones que caen entre estos parámetros, fue el que mejor comportamiento obtuvo en relación al objetivo perseguido. Del mismo modo, el operador DA agrega la Preferencia Parcial obtenida a partir de la agregación de las *Métricas de Código Fuente* con la Preferencia Elemental *Peso GUI*. Esto modela la siguiente situación: siempre que el método/función obtenga valores altos para las métricas del código fuente o interactúe con elementos de la GUI, lo que indica que podría ser un componente importante del sistema, ambas preferencias deben reemplazarse de alguna manera. El vector de peso es el resultado de establecer una prioridad mayor (2 sobre 1 según la Tabla 6.2) del *Peso GUI* sobre las *Métricas de Código Fuente*.

Por otro lado, las preferencias elementales *SUD* e *Intermediación* son agregadas por DA (función de Cuasi Disyunción media). Lo que se modela con este operador es que estas preferencias se reemplacen en cierta medida ya que brindan dos aspectos relevantes en relación a la importancia estructural del nodo, específicamente SUD (la misma es invertida por el Criterio Elemental[4]) permite identificar aquellos nodos que no son una utilidad bajo la perspectiva estructural que plantean sus autores, mientras que la *Intermediación* proporciona un valor que induce la importancia del artefacto desde un punto de vista topológico dentro del GELMF. Este operador está en el medio entre la disyunción pura y la media aritmética, por lo tanto los si alguna de las preferencias es bajo, no afecta en demasía el resultado final, a diferencia de los operadores de disyunción

[4] El valor es invertido para reflejar lo opuesto del Grado de Utilidad Estructural (SUD), específicamente $1 - SUD(r)$.

cercanos a la media aritmética La idea es que si alguna de las dos preferencias es alta, la misma pueda influenciar en mayor medida el la preferencia resultante. Por otra parte, en muchos casos la *Intermediación* resulta en 0 y si se considera un grado mayor de "conjuntividad" puede afectar de manera resonante en la evaluación de muchos nodos del GELMF. El esquema de ponderación es el resultado de priorizar la inversa de la métrica *SUD* (2 sobre 1 según la Tabla 6.2) ya que la misma brinda un aspecto más "local" de la evaluación del nodo, además el hecho que la *Intermediación* suele resultar en 0 en muchos nodos también refuerza dicho esquema.

Claramente, cuando un método/función sea un detalle de implementación, tiene que ser eliminado ya que como su denominación lo sugiere, no es importante. Es por este motivo que el resultado del operador DA, que agrega *SUD* e *Intermediación*, se agrega con la preferencia de *Detalle de Implementación*. El operador seleccionado es $C+-$ (Cuasi Conjunción fuerte) ya que se debe penalizar fuertemente los valores bajos de cualquier de estas dos preferencias, claramente se ha seleccionado un operador mandatorio (modela requerimientos obligatorios) ya que si alguna de las dos entradas es cero, el resultado del operador sea cero. En este sentido si el nodo es catalogado como un detalle de implementación o si el nodo es estructuralmente irrelevante, este operador penalizará fuertemente la Preferencia Global. El desequilibrio en el vector de ponderación está asociado a las preferencias y al tipo de operador; específicamente, la Preferencia Elemental correspondiente a la Variable de Preferencia *Detalle de Implementación* sólo puede ser 1 o 0, es decir que este criterio funciona a modo de "filtro" y por lo tanto debe ser agregado con un operador mandatorio y con un peso relativamente bajo. Es por este motivo que se priorizó en gran medida los valores correspondientes a la agregación de textitSUD e *Intermediación* (8 sobre 1 según la Tabla 6.2), esto resultó en el esquema de ponderación visualizado.

Finalmente, el operador $C-+$ agrega dos preferencias parciales que computan la Preferencia Global, la cual representará el valor IR para el nodo bajo evaluación. Este operador es una función de cuasi conjunción media que penaliza considerablemente los valores bajos para cualquier preferencia de entrada. Este tipo de operadores (intensidad media de cuasi-conjunción de polarización) se eligen con frecuencia para calcular la Preferencia Global (Miranda y cols., 2013; Su y cols., 1987; J. J. Dujmovic, 2007; Dujmović,

2013). Como se puede visualizar, el esquema de ponderación es equitativo ya que las dos preferencias son de igual importancia.

Como se mencionó en el análisis de cada operador, los pesos de las Preferencias Elementales y Parciales se definieron utilizando el enfoque *eigenvalue* que toma una comparación por pares como base para calcular los pesos de entrada para un operador según se explicó en la Subsección 6.4. Como ejemplo específico del cómputo se exhibe la estimación del peso para las *Métricas de Código Fuente*. La Tabla 6.3 muestra las puntuaciones relativas asignadas para las comparaciones por pares entre *Complejidad Ciclomática*, *Número de LOCs*, *Visibilidad* y *Tipo*. En la misma se puede destacar que los puntajes relativos han sido asignados en el rango comprendido entre 1 (*Importancia Equitativa*) y 3 (*Importancia Moderada*), esto significa que ningún criterio tiene valores de comparación extremos, es decir, ningún criterio es considerablemente más importante que los demás. La Complejidad Ciclomática (CC) ha sido considerada como el criterio más significativo seguida de la Cantidad de Lineas de Código ($LOCs$) para influir en la preferencia obtenida por el operador $D-$. En la Tabla 6.4 se muestra la matriz normalizada para el ejemplo expuesto anteriormente. El vector resultante proporciona un patrón del esquema de ponderación para los criterios que serán agregados por el operador bajo análisis

	CC	**#LOCs**	**Visibilidad**	**Tipo**
CC	1	2	3	3
#LOCs	1/2	1	3	3
Visibilidad	1/3	1/3	1	1
Tipo	1/3	1/3	1	1

Tabla 6.3: Matriz de comparación por pares para *Métricas de Código Fuente*

	CC	**#LOCs**	**Visibilidad**	**Tipo**	**Vec. prior.(%)**	**Vec. Peso(%)**
CC	6/13	6/11	3/8	3/8	**43,92**	**43**
#LOCs	3/13	3/11	3/8	3/8	**31,34**	**31**
Visibilidad	2/13	1/11	1/8	1/8	**12,37**	**13**
Tipo	2/13	1/11	1/8	1/8	**12,37**	**13**
Suma	1	1	1	1	**100**	**100**

Tabla 6.4: Matriz de comparación normalizada para los criterios agrupados en *Métricas de Código Fuente* y su correspondiente vector de peso

Finalmente, el vector de peso asignado para *Complejidad Ciclomática*, *Número de LOCs*, *Visibilidad* y *Tipo* es el siguiente $< 43, 31, 13, 13 >$ (ver Tabla 6.4). Al adoptar esta técnica para obtener un esquema de ponderación para cada operador, se pudo verificar la consistencia efectiva de los esquemas de ponderación obtenidos calculando la *tasa de consistencia (CR)* (Saaty, 2008).

Es importante resaltar que la Estructura de Agregación es uno de los elementos más importantes en el procedimiento de evaluación de LSP y que la definición de esta compleja estructura se estableció con la asistencia de ingenieros de software expertos en Ingeniería Reversa.

6.5. Proceso de Filtrado

El resultado final obtenido a través del método LSP en SAPSI es una Preferencia Global. Esta sugiere un grado de satisfacción del nodo con respecto a si el mismo es importante en el contexto de la lógica subyacente del sistema. Este grado de satisfacción está determinado por una métrica denominada IR, la cual puede tomar valores en el intervalo [0,1], donde los valores cercanos a 1 representan aquellos nodos que son relevantes y los valores próximos a 0 modelan lo contrario según la perspectiva que plantea la métrica.

Una vez que el valor de IR es estimado para todos los nodos en el GELMF, se debe

llevar a cabo un proceso de filtrado basado principalmente en esta métrica. Como muchos trabajos en Ingeniería de Software (Shawky y Abd-El-Hafiz, 2016; Elish y Al-Rahman Al-Khiaty, 2013; Chidamber y Kemerer, 1994; Hamou-Lhadj y Lethbridge, 2006), es importante establecer un *umbral* para el *grado de utilidad* al comienzo del proceso. Esto refleja cuándo un método/función se debe considerar como una *utilidad*. Por lo tanto, cada nodo cuyo valor de IR no supere dicho umbral se eliminará salvo por dos excepciones:

- que implemente el comportamiento de un componente interactivo de la GUI, es decir, si dicho nodo representa un método/función manejador de algún componente interactivo.
- que inicialice el sistema, particularmente aquellos métodos/funciones que crean y/o establecen conexiones iniciales con los componentes del contenedor principal de la GUI. Los mismos son considerados en un análisis posterior (ver apartado 7.2)

Esto se debe a que SAPSI considera a los manejadores como el componente esencial de todo el proceso de CP. Dichos nodos son extremadamente importantes ya que implementan funcionalidades que están altamente relacionadas con el Dominio del Problema.

El Algoritmo 2 presenta un pseudocódigo que describe cómo funciona el proceso de filtrado de nodos en el GELMF. Como se puede observar, el algoritmo recibe como entrada el GELMF, el umbral de utilidad y una opción (`iterativo`) que indica si se desea un filtrado iterativo. Si la última opción es falsa, el algoritmo realiza una sola iteración donde controla que cada nodo cumpla con las condiciones para ser filtrado, específicamente que no supere el umbral de utilidad (`umbralUt`), que no sea manejador (`!nodo.esManejador()`) ni tampoco inicializador (`!nodo.esInicializador()`). La función `filtrarNodo` recibe el nodo a filtrar junto con el grafo y efectúa la eliminación al mismo tiempo que actualiza los datos de la representación (por ejemplo, arcos de entrada y de salida en el grafo). Finalmente, la función `eliminarComponentesAislados` busca nodos (o grupo de nodos) sin arcos de entrada o salida y los filtra. Estos nodos deben cumplir la misma condición que requiere `filtrarNodos` para filtrar los nodos, es decir no deben ser ni manejadores ni inicializadores. La función calcula el cierre transitivo para cada nodo marcado como "no manejador" y "no inicializador" en la fase de extracción

de información y luego filtra todos los nodos restantes que no se incluyen en este conjunto. Si esto no ocurriera, estos nodos obstaculizarían el análisis posterior ya que están aislados de los nodos restantes en GELMF.

```
input : grafo // GELMF atribuido con todas las métricas.
input : umbralUt // valor umbral de utilidad.
input : iterativo // indica si el filtrado es iterativo (valor booleano).
output: GELMF' // grafo sin utilidades.
Data: cantNodos // número de nodos eliminados en cada iteración.
1  do
2      cantNodos=0;
3      foreach nodo ∈ grafo do
4          if nodo.obtenerIR() ≤ umbralUt and
5          !nodo.esManejador() and !nodo.esInicializador() then
6              filtrarNodo(nodo,grafo);
7              cantNodos++;
8          end
9      end
10     if iterativo then
11         ajustarSUDyBC(grafo);
12     end
13     else
14         canNodos=0;
15     end
16 while cantNodos != 0;
17 eliminarComponentesAislados(grafo);
```

Algoritmo 2: Algoritmo de reducción de información

Si la opción `iterativo` es verdadera, funciona de la misma manera sólo que realiza más iteraciones hasta que no haya más eliminaciones entre iteraciones. En este caso el factor de modificación entre iteraciones es la función `ajustarSUDyBC`, que vuelve a computar las métricas *Grado de Utilidad Estructural (SUD)* e *Intermediación (BC)* en conjunto con IR para cada nodo restante en `GELMF'`. Esta variante se presenta para ser usado con sistemas grandes en situaciones donde se requiera una reducción más sustancial de la información del GELMF. Por otra parte si el sistema es de tamaño pequeño/mediano, sólo basta con realizar una iteración de filtrado. El criterio de selección del valor para la variable `iterativo` está relacionado con el tamaño del sistema y la cantidad de información que se desee filtrar.

El resultado del algoritmo es la representación de base que utiliza SAPSI, es decir el

GELMF, conteniendo aquellos nodos relevantes para la lógica subyacente del sistema los cuales son esenciales para el objetivo principal perseguido por la estrategia de CP.

Es importante remarcar que el valor de `umbralUt` debe ser especificado como un argumento al inicio del proceso de la estrategia. Esta característica permite al ingeniero de software definir un umbral teniendo en cuenta las siguientes consideraciones:

- Las características que presenta el sistema bajo estudio, como por ejemplo lenguaje/s de programación, estimaciones del tamaño del sistema (por ejemplo, número de archivos o #LOCs), herramientas externas empleadas y/o paradigmas de programación utilizados.
- Experiencia con la estrategia SAPSI.
- Análisis de los resultados usando diferentes rangos de umbrales para el mismo sistema.

En la Sección A.2 del Anexo B se detalla el proceso completo de reducción de información para el ejemplo de la libreta de contactos desarrollada en Python. En el mismo también se analizan los distintos factores que afectan el proceso de filtrado y ciertas características que deben ser tenidas en cuenta para la selección del valor `umbralUt`.

6.6. Notas y Comentarios del Capítulo

En este capítulo se presentó el proceso general para reducir el tamaño de la representación de base que usa SAPSI para el análisis. Este proceso de reducción de información se basa en el cálculo de una métrica denominada Importancia Relativa (IR) cuyo valor representa un grado de certeza respecto de si el nodo es relevante o no en relación a la lógica subyacente del sistema.

El enfoque adoptado por SAPSI es frecuentemente utilizado en el contexto de Ingeniería Reversa y requiere de la definición de un umbral que permita determinar cuando un método/función es irrelevante para el análisis. Sin embargo, no se han encontrado trabajos en la literatura referente que permitan integrar distintos aspectos de una artefacto de software en particular en una única métrica de software.

SAPSI posibilita el cálculo de la métrica IR para cada método/función tomando como base distintos requerimientos que el mismo debe cumplir para ser calificado como importante. Para el cálculo hace uso de un MEMC denominado LSP, el cual requiere la especificación de:

- un Árbol de Criterios que contenga los requerimientos a cumplir por parte de cada elemento bajo análisis,
- un conjunto de funciones de Criterio Elemental que normalizan los diversos valores que pueden asignarse a los distintos requerimientos en el árbol,
- una Estructura de Agregación que posibilita la agregación en distintos niveles de las Preferencias hasta obtener la Preferencia Global que indica el grado de satisfacción de los aspectos que se están evaluando para un elemento en particular.

Todo el proceso decanta en la obtención de un valor que representa el grado de satisfacción de un nodo (método/función) del GELMF en referencia la *Importancia Relativa* que el mismo posee respecto de la lógica subyacente del sistema. Finalmente, tomando como base los valores de IR de cada nodo y un umbral de filtrado, se lleva a cabo un proceso específico que elimina todos los nodos que cumplen con ciertas características y además con valores de IR que no superan el umbral. Se proponen dos tipos de algoritmos, uno para sistemas medianos/pequeños que realiza una sola pasada y otro para sistemas grandes lleva a cabo un proceso iterativo. El resultado del filtrado es un GELMF que contiene sólo aquellos nodos relevantes para la lógica subyacente del sistema y, por consiguiente, esenciales para llevar a cabo el análisis propuesto por SAPSI.

Es importante mencionar que la flexibilidad de LSP y el proceso de filtrado hacen evidente la posibilidad de calibrar todo el modelo de acuerdo a nuevos requerimientos ya sea porque fueron detectados durante el proceso de Ingeniería Reversa o porque fueron impuestos por el mismo. Claramente, esto se puede hacer ajustando las funciones de criterios elementales (cambiando x_{max}, x_{min} o tipos de funciones) y ajustando pesos y operaciones en la estructura de agregación. La flexibilidad del método permite que este procedimiento de calibración sea conducido principalmente por la experiencia del ingeniero de software.

Capítulo 7

Técnica de Clustering y Generación del Modelo de Casos de Uso

Todas las piezas deben unirse sin ser forzadas. Debe recordar que los componentes que está ensamblando fueron desmontados por usted, por lo que si no puede unirlos debe existir una razón. Pero sobre todo, no use un martillo.

— Manual de mantenimiento de IBM, año 1925

Aunque el filtrado de información irrelevante ayuda a reducir información en cualquier contexto, las representaciones extraídas por la estrategia pueden llegar a alcanzar tamaños considerables en sistemas grandes. Aún teniendo en cuenta sistemas relativamente pequeños, la extracción de información suele ser difícil de visualizar, incluso cuando se reduce información en grandes proporciones (Miranda y cols., 2015). Es por esto que es necesario un mecanismo de abstracción más sólido, que posibilite no sólo la reducción del tamaño de las estructuras sino que brinde abstracciones para facilitar el entendimiento de las mismas. En este sentido, es posible caracterizar a las técnicas de clustering como unos de los mecanismos más utilizados en el contexto de Ingeniería Reversa para agrupar y abstraer artefactos de un sistema de software. Las técnicas de clustering permiten agrupar entidades de un sistema, tales como clases, funciones o ar-

chivos, en componentes significativas (por ejemplo, subsistemas) con el objetivo de asistir en la comprensión de la estructura de un sistema de gran envergadura.

En esta etapa SAPSI propone una técnica de clustering cuasi-automática que agrupa los nodos del GELMF progresivamente tomando como principal criterio de agrupamiento los nodos que están estrechamente vinculados con la GUI del sistema para luego plasmar un modelo de casos de uso. Finalmente, se expone el algoritmo que mapea el modelo de cluster[1] en un modelo de casos de uso de UML.

El capítulo se organiza de la siguiente manera, en primer lugar se introduce el concepto de clustering, los tipos de algoritmos y sus características (Sección 7.1); posteriormente se expone el algoritmo de clustering desarrollado por SAPSI (Sección 7.2); luego se explica la construcción de un modelo de casos de uso de UML a partir del modelo de cluster obtenido mediante el algoritmo de clustering (Sección 7.3); finalmente se presentan las notas y comentarios del capítulo (Sección 7.4).

7.1. Técnicas de Clustering de Software

En general, las técnicas de clustering (o agrupamiento) se pueden caracterizar como un conjunto de técnicas multivariadas cuyo propósito principal es agrupar entidades en función de atributos que caracterizan a las mismas. Las mismas entidades se clasifican de acuerdo a ciertos criterios de análisis predeterminados con el objetivo de que entidades estrechamente relacionadas sean agrupadas en el mismo cluster. El objetivo de cualquier algoritmo de clustering es ordenar las entidades en grupos, de modo que la variación entre clusters se maximice en relación con la variación dentro de los mismos.

Desde el punto de vista del software, las técnicas de clustering permiten agrupar entidades de un sistema de software incluyendo métodos, clases, archivos o cualquier artefacto de software en particular, en subsistemas significativos para asistir durante el proceso de comprensión de un sistema de software de tamaño considerable (Maqbool y Babri, 2007; Tzerpos y Holt, 2000). Un enfoque exitoso para llevar a cabo esta tarea puede tener un valor práctico significativo para los ingenieros de software, particularmente aquellos que trabajan en sistemas legados donde la mayoría de los casos la documentación

[1]La agregación obtenida a partir de ejecutar el algoritmo de clustering sobre el GELM

es obsoleta o inexistente.

Si bien es posible hacer referencia a diversos aspectos relativos las técnicas de clustering, en los siguientes apartados se detallan aquellos que son más relevantes dentro del contexto de este trabajo.

7.1.1. Medidas de Similaridad

En muchos trabajos el principal criterio de agrupamiento es el uso de medidas de similaridad, es decir una función que permite cuantificar la similitud entre dos objetos. Por lo general, las técnicas de clustering históricamente han empleado tres tipos de medidas de similaridad para determinar las entidades que serán agrupadas en un mismo cluster, a saber, medidas de distancia, coeficientes de correlación y coeficientes de asociación. Las medidas de distancia describen numéricamente qué tan separadas están las entidades teniendo en cuenta un espacio y normalmente son utilizadas cuando las variables que representan las características son de carácter continuo (Mitchell y Mancoridis, 2001).

Por lo general, los coeficientes de correlación se usan para correlacionar variables continuas, mientras que los coeficientes de asociación se suelen aplicar a variables binarias. En las Tablas 7.1 y 7.2 se exhiben algunas medidas de distancia y similaridad conocidas en el contexto de clustering.

Medida de distancia	Fórmula
Distancia de Minkowski	$(\sum_{i=0}^{s} \lvert x_i - y_i \rvert^r)^{1/r}$
Distancia de Canberra	$\frac{\sum_{i=0}^{s} \lvert x_i - y_i \rvert}{(\lvert x_i \rvert + \lvert y_i \rvert)}$
Distancia de Bray-Curtis	$\frac{\sum_{i=0}^{s} \lvert x_i - y_i \rvert}{\sum_{i=0}^{s} (x_i + y_i)}$
Distancia de Chord	$\sqrt{2\left(1 - \frac{\sum_{i=0}^{s} x_i y_i}{\sqrt{\sum_{i=0}^{s} x_i^2 \sum_{i=0}^{s} y_i^2}}\right)}$

Tabla 7.1: Medidas de distancia. Para la distancia de Minkowski, $r = 1$ representa la distancia de ciudad o de *Manhattan* mientras que $r = 2$ representa la distancia *Euclideana*.

Medida de similaridad	Fórmula
Coeficiente Jaccard	$\frac{a}{a+b+c}$
Sorenson-Dice	$\frac{2a}{2a+b+c}$
Rogers y Tanimoto	$\frac{a+d}{a+2(b+c)+d}$
Sokal y Sneath	$\frac{a}{a+2(b+c)}$

Tabla 7.2: Coeficientes de asociación.

En la Tabla 7.1, x e y representan puntos en el espacio Euclidiano R^s. En la Tabla 7.2, a representa el número de características que coinciden en ambas entidades, d representa el número de características que están ausentes en ambas entidades, mientras que b y c representan las características que están presentes para una entidad pero no para la otra y viceversa.

Los algoritmos de clustering que usan estas medidas agrupan las entidades siguiendo algún criterio basado en las mismas. Por ejemplo, los algoritmos aglomerativos calculan las distancias entre todas las entidades y posteriormente observan cuáles son las más próximas en cuanto a esta distancia o similaridad. Estas formarán un grupo que no vuelve a separarse durante el proceso. Se mide la distancia o similaridad entre todas entidades de nuevo, tomando el grupo ya formado como si se tratara de una unidad. Existen distintas variantes del algoritmo de acuerdo a la forma de calcular la distancia o similaridad entre todas las entidades restantes y el grupo nuevo. El lector interesado en analizar este tipo de algoritmos puede leer la bibliografía referente al tema (Anquetil y Lethbridge, 1999; Mitchell y Mancoridis, 2001).

Por otra parte, también se han definido criterios de agrupamiento específicos para clustering de artefactos de software. Schwanke (Schwanke, 1991) introdujo la noción de utilizar principios de diseño, como bajo acoplamiento y alta cohesión. Por su parte Koschke (Koschke, 2002) extendió la técnica de Schwanke definiendo métricas que además de los aspectos de diseño también consideran otro tipo de información como declaraciones globales de variables, llamadas a funciones, similitudes entre identificadores de funciones y archivos, entre otros. En el Capítulo 3 se presentan varios trabajos que extienden este

tipo de enfoques haciendo uso de métricas de software.

7.1.2. Creación de Clusters

Una vez establecido el criterio de agrupamiento, existen diversas formas en las que la técnica aglomera las entidades y define los distintos clusters. En este apartado se describen aquellas técnicas más conocidas dentro de este contexto (Wiggerts, 1997).

Algoritmo Basados en Teoría de Grafos

Esta clase de algoritmos se basa en propiedades inherentes a los grafos. En dichos grafos, los nodos representan entidades, y los arcos representan relaciones entre las mismas. La idea principal es identificar subgrafos interesantes que se usarán como "puntos de partida" para formar los clusters. Los tipos de subgrafos que se pueden usar para este fin incluyen componentes conectados, *cliques* y *spanning trees*. Los dos tipos más comunes de algoritmos de agrupamiento teórico de gráficos son los *algoritmos de agregación* y los algoritmos de árbol de cobertura mínima (*Minimum Spanning Tree* (MST)).

Los algoritmos de agregación reducen la densidad de nodos en el grafo al fusionarlos en clusters tomando como base algún criterio en particular. De esta manera se obtiene un grafo reducido donde cada cluster representa de alguna manera al grupo de nodos que lo compone. Además es posible que los clusters formen parte de nuevas iteraciones y de esta manera se forma una jerarquía con distintos niveles de agregaciones. Como ejemplo de este tipo específico de técnicas de reducción de grafos se pueden destacar la noción de *vecindario* de un nodo (von Laszewski, 1993), componentes fuertemente conectados y bicomponentes (Botafogo y Shneiderman, 1991) entre otros (Pothen, 1997).

Los algoritmos de árbol de cobertura mínima toman como base la estructura MST y luego definen heurísticas para tratar con aquellos casos que no son resueltos usando la estructura inicial (Bauer y Trifu, 2004).

Algoritmos *Constructivos*

Los algoritmos en esta categoría asignan las entidades a los clusters en una sola pasada. Los clusters pueden estar predefinidos de entrada (supervisados) o ser construidos

como parte del proceso de asignación (sin supervisión). Los ejemplos de algoritmos de construcción más conocidos incluyen las llamadas técnicas geográficas y las técnicas de búsqueda por densidad. Las primeras son referenciadas de esta forma ya que requieren que las entidades a ser analizadas estén "organizadas", en cierto modo, en un espacio bidimensional como en los mapas. Una técnica geográfica bien conocida es el algoritmo de bisección, que en cada paso divide el plano en dos y agrupa cada entidad de acuerdo al sector del plano en el que se encuentra la misma.

Por otra parte, los algoritmos basados en búsqueda por densidad intentan identificar regiones densas del grafo para así conformar los clusters que son el punto de partida para los demás agrupamientos. Los clusters pueden ser fusionados para formar nuevos clusters y los nodos pueden ser asignados a clusters ya existentes. El algoritmo avanza hasta obtener el modelo de cluster final, factor determinado por un criterio de parada. Wishert (WISHERT, 1969) presenta una técnica que utiliza búsqueda por densidad denominada *mode analysis*. La misma analiza cada nodo del grafo asociando al mismo los vecinos dentro de un radio r. A partir de esto, toda entidad que esté rodeada por más de k vecinos es considerada como *densa* y conforma un nuevo cluster.

Algoritmos Basados en Optimización

Un algoritmo de optimización toma una solución inicial e intenta mejorar esta solución mediante adaptaciones iterativas en función de alguna heurística. El método de optimización se utiliza para generar agrupamientos jerárquicos (Lutz, 2001) y no jerárquicos (Mamaghani y Meybodi, 2009).

Un método de clustering mediante optimización no jerárquico típico comienza con una partición inicial la cual es derivada en base a alguna heurística. Luego, las entidades se mueven a otros clusters para mejorar la partición de acuerdo con algunos criterios. Esta reubicación continúa hasta que no se produce ninguna mejora adicional de este criterio. Algunos ejemplos de métodos de clustering que usan optimización son presentados por Clarke y su grupo de investigación (Clarke y cols., 2003).

En términos generales, los algoritmos de optimización se pueden clasificar en dos categorías que se describen brevemente a continuación:

- *Algoritmos genéticos:* son técnicas de búsqueda y optimización aleatorias GUIadas por los principios de la evolución y la genética natural, que tienen una gran cantidad de paralelismo implícito. Los algoritmos genéticos se caracterizan por ciertos atributos, como la función objetivo, la codificación de los datos de entrada, los operadores genéticos, como el cruce y la mutación, y el tamaño de la población. Generalmente, un algoritmo genético de clustering se comporta de la siguiente manera: i) selecciona una población aleatoria de particiones; ii) genera una nueva población seleccionando los mejores individuos de acuerdo con la función objetivo y reproduciendo los nuevos mediante el uso de las operaciones genéticas; iii) repite el paso ii) hasta que se satisfaga un criterio de parada predefinido.

- *Hill-climbing:* Los métodos de búsqueda de *hill-climbing* se han empleado con éxito en varios algoritmos de clustering de software. Mancoridis y Mitchell (Mancoridis y cols., 1998, 1999) muestran resultados prometedores en términos de la calidad y el rendimiento de los métodos de búsqueda de *hill-climbing*. Su enfoque ha sido implementado como parte de la herramienta de clustering de software *Bunch* (Mancoridis y cols., 1999). La herramienta comienza generando una partición aleatoria del grafo de dependencia de módulos. Posteriormente, las entidades de la partición se reagrupan sistemáticamente al examinar las particiones vecinas para encontrar una mejor. Cuando se encuentra una partición mejorada, el proceso se repite; es decir, la partición encontrada se usa como base para encontrar otras mejores. El algoritmo se detiene cuando no puede encontrar una mejor partición. La función objetivo es la función de calidad de modularización utilizada también por el algoritmo genético de *Bunch*.

Algoritmos jerárquicos

Es posible distinguir dos categorías de algoritmos jerárquicos: divisivo (*top-down*) y aglomerativo (*bottom-up*). Los algoritmos divisivos comienzan con un cluster que contiene todas las entidades y divide el mismo en un número (generalmente dos) de clusters separados en cada paso sucesivo. Los algoritmos aglomerativos comienzan en la parte inferior de la jerarquía agrupando iterativamente entidades similares formando de esta

manera los clusters. En cada paso, los dos clusters más similares entre sí se fusionan, y el número de clusters se reduce en uno.

Los algoritmos divisivos ofrecen una ventaja sobre los algoritmos de aglomeración porque la mayoría de los usuarios están interesados en la estructura principal de los datos, que consiste en unos pocos grupos grandes que se encuentran en los primeros pasos de los algoritmos divisivos. Los algoritmos de aglomeración comienzan con entidades individuales y se abren paso hacia grandes grupos que pueden verse afectados por decisiones desafortunadas en los primeros pasos. Sin embargo, los algoritmos jerárquicos aglomerativos son los más utilizados; esto se debe a que es inviable considerar todas las divisiones posibles de los primeros grupos grandes (Wiggerts, 1997).

En términos generales, los algoritmos aglomerativos ejecutan las siguientes tareas: i) calcula una matriz de similaridad; ii) encuentra los dos grupos más similares y los agrupa; iii) calcula la similaridad entre los clusters unidos y algunos otros obteniendo una matriz reducida; y iv) repite desde el paso ii) hasta que solo queden dos grupos.

Andritsos y Tzerpos (Andritsos y Tzerpos, 2005) presentaron el algoritmo *Scalable Information Bottleneck (LIMBO)*, un algoritmo jerárquico aglomerativo que emplea la técnica de cuello de botella para la agrupación. LIMBO utiliza una medida la cual toma como principal criterio minimizar la pérdida de información para calcular la similaridad entre entidades. En cada paso, se elige el par de entidades que daría como resultado la menor pérdida de información. ACDC (Tzerpos y Holt, 2000) es un algoritmo de agrupamiento jerárquico que no sigue un esquema estándar. No se puede asignar a la categoría aglomerativa o divisiva porque el algoritmo no tiene una división explícita iterativa o una etapa de fusión. ACDC usa una estrategia de detección de ciertos patrones los cuales han demostrado tener buenos resultados sobre la descomposición final, sobre todo a la hora de llevar a cabo tareas de CP (Garcia y cols., 2013). El algoritmo aplica sistemáticamente estos patrones de subsistema a la estructura del software. Esto resulta en que la mayoría de los módulos se colocan en categorías jerárquicas (subsistemas). Además de esto, ACDC propone un algoritmo de *adopción de huérfano*[2] para asignar los módulos restantes al subsistema apropiado.

[2]Técnica utilizada para determinar la pertenencia de "nodos bamboleantes" (*huérfanos*) a clusters existentes. Los *huérfanos* representan nuevos recursos del sistema producto de las tareas de mantenimiento y la natural evolución del sistema.

Es importante resaltar que ACDC es uno de los primeros algoritmos que propuso un enfoque orientado a la CP y es por este motivo que se ha tomado como base para otros trabajos enmarcados en la misma temática (Bauer y Trifu, 2004; Garcia y cols., 2013; Maqbool y Babri, 2007).

Hasta el momento se han discutido varios algoritmos y técnicas de similaridad que se han adaptado al contexto de clustering del software. Una observación importante es que existen dos enfoques conceptuales diferentes para desarrollar una metodología de clustering en este contexto. El primero intenta desarrollar un enfoque de descubrimiento sofisticado de estructuras como *ACDC* y *Bunch*. El segundo enfoque se concentra más en desarrollar funciones de similaridad (Schwanke, 1991; Koschke y Eisenbarth, 2000).

7.1.3. Rotulado de Clusters

Más allá de la heurística en la cual esté basado el algoritmo de clustering de software, no debe perderse de vista el principal objetivo del mismo: el modelo de cluster resultante debe facilitar al ingeniero de software la compresión del sistema bajo análisis. Este principal objetivo será difícil de alcanzar si los clusters no están rotulados de manera apropiada, aún cuando la descomposición obtenida sea la mejor. Asignar rótulos significativos al modelo de clusters posibilitará una interpretación más clara del agrupamiento resultante. Uno de los primeros trabajos sobre rotulado de clusters es de Schwanke y Platoff (Schwanke y Platoff, 1993), quienes utilizaron un resumen de características para sugerir término adecuados que luego serían asignados de forma manual. Tzerpos (Tzerpos y Holt, 2000) utiliza un enfoque basado en patrones para reconocer estructuras de subsistemas familiares dentro de sistemas grandes y rotularlos automáticamente. Tonella y su grupo de investigación (Tonella y cols., 2003) definen los nombres de los clusters en base a palabras clave encontradas dentro la página web analizada. A medida que avanza el proceso de agrupamiento, los clusters son rotulados utilizando un enfoque de frecuencia inversa sobre un conjunto de términos posibles. Todos los términos asociados a cada entidad en el cluster forma parte de una base sobre la cual se realizarán consultas.

Finalmente, cada cluster es etiquetado con los términos que figuran en el top 10 de las consultas realizadas. Maqbool y Babri (Maqbool y Babri, 2005, 2006) emplean los identificadores de las funciones que integran cada cluster como palabras claves representativas en el rótulo de los mismos.

Si bien es posible encontrar distintos enfoques, es importante destacar que la estrategia definida para nombrar los clusters cumple un rol esencial dentro del proceso general. Como bien afirman ciertos autores, una estrategia débil de definición de nombres, puede hacer fracasar la estrategia de clustering (Tonella y cols., 2003; Tzerpos y Holt, 2000; Shtern y Tzerpos, 2012).

7.1.4. Medidas de Evaluación

Los investigadores han desarrollado varios métodos para evaluar de alguna manera las descomposiciones obtenidas mediante algoritmos de clustering software. Dicho aspecto es relevante ya que:

- La evaluación ayuda a descubrir las distintas fortalezas y debilidades de los algoritmos de clustering. Esto posibilita el análisis de las características principales destacando las situaciones en donde es conveniente aplicarlo y por otra parte, atenuar las debilidades encontradas.

- La evaluación puede determinar el dominio de aplicación de cada algoritmo, identificado los contextos adecuados donde se puede aplicar cada uno.

Básicamente, los enfoques para evaluación de algoritmos de clustering de software pueden ser clasificados en dos categorías, dependiendo de si se basan en la existencia de una descomposición *autoritativa*, o no. Este es un modelo del sistema generado por un experto, la cual refleja una descomposición en subsistemas significativos del mismo. Por el contrario, los modelos de clusters creados mediante algoritmos de clustering de software son obtenidos de manera automática (o cuasi-automática).

En términos generales, es posible distinguir dos criterios de evaluación que no están basados en descomposiciones autoritativas: *estabilidad* y *extremidad de la distribución de los clusters*. El primero refleja cuan sensible es una técnica de clustering a las modi-

ficaciones de los datos de entrada. El razonamiento detrás de la estabilidad en una descomposición es que para versiones similares de un sistema de software se deben producir descomposiciones similares. Una función de estabilidad mide el porcentaje de cambios entre las descomposiciones producidas de las versiones sucesivas de un sistema de software en evolución. Teniendo pequeños cambios entre versiones consecutivas, un algoritmo debería producir un agrupamiento similar. El segundo se basa en el análisis del tamaño de los clusters producidos por la técnica de clustering. El razonamiento básico detrás de este criterio de evaluación es medir el balance general del modelo con respecto a los tamaños que exhiben los clusters. En otras palabras, un algoritmo de clustering debe evitar las siguientes situaciones: i) la mayoría de las entidades están agrupados en uno o pocos grupos grandes, ii) la mayoría de los clusters están conformados por una única entidad. El principal inconveniente que presentan ambos criterios es que sólo permiten identificar "malas" descomposiciones de software, de acuerdo a métricas impuestas por algunos autores. Sin embargo, no permiten determinar si la descomposición efectivamente refleja una abstracción del sistema. En este sentido, es fácil desarrollar un algoritmo que produzca descomposiciones que sean estables y no extremas, y que sea completamente inútil para mejorar la comprensión del sistema bajo análisis (Tzerpos y Holt, 2000; Shtern y Tzerpos, 2012).

Por otra parte, ciertos autores han propuesto métodos de evaluación basados en descomposiciones autoritativas. En este sentido es posible identificar las siguientes como las más utilizadas en los trabajos de investigación:

- *Familia de medidas de distancia MoJo*: Tzerpos y Holt desarrollaron una medida de distancia llamada MoJo (Tzerpos y Holt, 1999). La misma intenta capturar la distancia entre dos descomposiciones como el número mínimo de operaciones *Move* y *Join* que se deben realizar para transformar una descomposición en la otra. Una operación *Move* implica la reubicación de una única entidad de un cluster a otro (o a un nuevo cluster), mientras que una operación *Join* toma dos clusters y los fusiona en un único cluster. La medida MoJo dio lugar a una familia de medidas que con el tiempo fueron mejorando la original, por ejemplo, calculando la medida en tiempo polinomial y además produciendo un valor en el intervalo [0,100] (Wen y Tzerpos, 2004); haciéndola apta para medir descomposiciones anidadas (Shtern

y Tzerpos, 2007); entre otras.

- *Precision/Recall*: *Precisión* y *recuperación* son métricas estándares de recuperación de información. Se han usado para evaluar clustering de software por Anquetil y Lethbridge (Anquetil y Lethbridge, 1999). El método calcula similaridades basadas en la medición de *intra-pares*, que son pares de entidades que pertenecen al mismo cluster. Los autores sugieren calcular la precisión y la recuperación para una partición dada de la siguiente manera: i) precisión: porcentaje de intra-pares propuestos por el método de clustering, que también son intra-pares en la descomposición autoritativa. ii) recuperación: porcentaje de intra-pares en la descomposición autoritativa, que también son intra-pares en la descomposición propuesta por el método. Un aspecto indeseable de la precisión/recuperación es que el valor de esta medida es sensible al tamaño y al número de clusters (Mancoridis y cols., 1999).

- *Medida propuesta por Koschke y Eisenbarth*: Koschke y Eisenbarth (Koschke y Eisenbarth, 2000) presentaron una forma de comparar cuantitativamente las particiones automáticas con las autoritativas estableciendo una escala donde miden cuan buenas son las correspondencias entre el modelo de cluster y la descomposición autoritativa. En términos generales, el procedimiento plantea los siguientes pasos: i) identificar los grupos que se corresponden de buena manera, es decir que tienen un buen porcentaje de coincidencias entre sí (coincidencias calificadas como *good match*); ii) identificar subclusters correspondientes, es decir, donde alguna parte de un cluster corresponde a una parte de otro cluster (coincidencias calificadas como *OK match*); y iii) medir la precisión de las correspondencias entre ambas dos particiones. Esta medida de evaluación posee dos inconvenientes: en primer lugar no penaliza el algoritmo de agrupamiento de software cuando una descomposición automática es más detallada que la autorizada, y en segundo lugar la tasa de *recuperación* no distingue entre *good match* y *OK match*. Los autores también han desarrollado escenarios de referencia para la evaluación de clustering de software y la calibración de parámetros de clustering de software.

Si bien los métodos de comparación basados en descomposiciones autoritativas es

uno de los métodos más utilizados por los investigadores, también poseen la desventaja de que una descomposición se puede construirse de muchas maneras válidas siguiendo distintos criterios. Es decir, puede que dos expertos con el mismo nivel de conocimientos sobre el sistema obtengan modelos correctos, pero diferentes. Por lo tanto, si la descomposición es construida con un objetivo distinto que el del algoritmo de clustering, los resultados de la comparación seguramente se verán afectados por este factor. Sin embargo, hasta el momento no se conoce un criterio de comparación más confiable que este (Shtern y Tzerpos, 2011; Wen y Tzerpos, 2004). Una forma de reducir este tipo de imprecisiones es definir distintas formas de interactuar con los expertos que generaron las descomposiciones y de esta manera poder interpretar mejor los resultados obtenidos por el algoritmo.

En lo que resta de este capítulo se presenta la técnica de clustering implementada por SAPSI para obtener un modelo de clusters y posteriormente se detalla el algoritmo que permite obtener un modelo de casos de uso de UML para el sistema bajo estudio.

7.2. Técnica de Clustering de SAPSI

En esta etapa la estrategia implementa una técnica de clustering la cual es primordial con respecto a la abstracción e identificación de las funcionalidades elementales del sistema bajo estudio. Con este propósito la estrategia define un técnica de clustering aglomerativa que va agrupando las entidades del GELMF formado de esta manera un modelo de cluster del sistema. Para esto se toma como base información extraída en etapas previas. Esta información refleja distintos aspectos del sistema analizando tales como los componentes de la GUI, el rol topológico que cumple cada nodo en el GELMF, entre otros. En este sentido, algunos autores han manifestado que al incrementarse el número de características significativas analizadas por la estrategia de clustering, los resultados obtenidos se acercan más al modelo obtenido por los expertos (Maqbool y Babri, 2007; Anquetil y Lethbridge, 1999). Por otra parte, uno de los criterios principales que dirige al algoritmo de clustering propuesto está relacionado con los componentes de la interfaz gráfica. Estos cumplen un rol fundamental en el sistema y están estrechamente relacionados con el Dominio del Problema (Memon y cols., 2003; Almendros-Jimenez y Iribarne,

2005). El algoritmo realiza un proceso iterativo donde aquellos artefactos de software que son más relevantes para la lógica subyacente del sistema, comienzan a aglomerarse con los demás definiendo los clusters más importantes y que determinan el punto de partida del algoritmo.

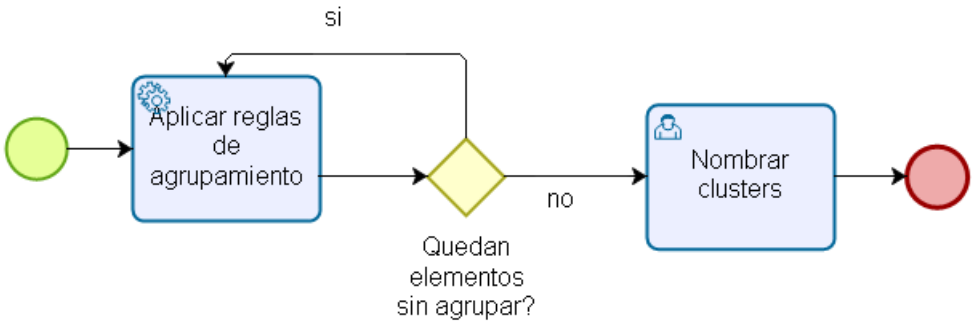

Figura 7.1: BPMN correspondiente a la etapa Construcción de Modelo de Clusters.

En la Figura 7.1 se muestra las actividades relacionadas con el proceso descripto en esta sección. El mismo recibe como entrada el GELMF resumido que devuelve la etapa anterior donde cada nodo contiene información que describe el método/función que el mismo representa. Específicamente, en este punto, cada nodo del grafo está atribuido con las métricas de software extraídas y la métrica IR.

En los siguientes apartados se explican las etapas que comprende la técnica de clustering propuesta por SAPSI hasta llegar a un modelo de cluster el cual consta específicamente de clusters, y relaciones entre los mismos.

7.2.1. Etapa 1: Detección de Nodos Inicializadores

Se identifican todos los nodos que representan métodos/funciones que inicializan el sistema, particularmente aquellos que crean y/o establecen conexiones iniciales con los componentes del contenedor principal de la GUI. Estos nodos están agrupados en un nuevo cluster etiquetado como 'INIT'. En general, este tipo de nodos están frecuentemente relacionados con los componentes de la GUI; de hecho, en diversos sistemas, los mismos llevan a cabo una fuerte carga computacional. Por lo tanto, es una buena práctica detectarlos y visualizarlos con el propósito de ayudar al ingeniero de software a analizar este tipo de artefactos y el papel que juegan dentro del sistema.

7.2.2. Etapa 2: Análisis de Manejadores

Cada componente de GUI interactivo (como botones, listas desplegables y elementos de menú) tiene un método o función manejador (o controlador) del código fuente que implementa su comportamiento. Los componentes interactivos de la GUI se pueden vincular con los casos de uso porque están estrechamente relacionados con la interacción del usuario (actor) (Bojic y Velasevic, 2000; Almendros-Jimenez y Iribarne, 2005). SAPSI define un nuevo cluster para cada uno de estos controladores de componentes de la GUI, este tipo de clusters es referenciado a lo largo de este trabajo como *cluster manejador* o *cluster controlador*. Además, también son agrupados los nodos que son llamados exclusivamente por aquellos métodos/funciones que dieron origen al cluster manejador cluster. Esta dependencia es directa y trivial ya que las tareas llevadas a cabo por estos nodos sólo son invocadas únicamente por los manejadores.

Cada cluster manejador es rotulado inicialmente usando algún término extraído del componente GUI asociado al manejador. Este término se busca en el ARGUI realizando un recorrido en profundidad desde ese componente de la GUI. El algoritmo busca aquellos widgets que contengan cualquier término relacionado que pueda rotular el cluster (por ejemplo, rótulo del botón, nombre de archivo de la imagen o títulos en los bordes de los componentes). En algunas situaciones, este algoritmo puede no proporcionar ninguna denominación apropiada para el cluster en cuestión. En este caso, la etiqueta es determinada por el ingeniero de software en una de las etapas subsiguientes (etapa 4).

7.2.3. Etapa 3: Agrupamiento Estructural

Esta etapa analiza todos los nodos que no se han agrupado hasta esta instancia del proceso. En este contexto, se han definido ciertas reglas que se basan en la definición de un valor umbral de importancia ($umbralIm$) que representa cuando un nodo o cluster se debe considerar como importante para el sistema tomando como base la métrica IR. De manera similar al umbral para la detección de utilidad ($umbralUt$), el valor para $umbralIm$ se debe proporcionar como entrada al comienzo del proceso. Estas reglas se pueden clasificar teniendo en cuenta dos criterios:

Agregar nuevos elementos a clusters ya definidos

Se define un conjunto de reglas que establece cuándo un nodo (o cluster) debería unirse a otro cluster predefinido. Tres de las más relevantes son:

i. Si sólo un cluster (*Cluster A*) depende de un único nodo (o *viceversa*), este último se agrupa en el cluster anterior. Tal es el caso de un conjunto de métodos/funciones que utiliza exclusivamente un método/función particular. Por lo tanto, si no hay otro nodo o cluster dependiendo de la ejecución de este último, entonces debe agruparse con el grupo de nodos que emplea exclusivamente ese nodo. En la Figura 7.2 **a** se representa la situación contemplada por esta regla.

ii. Si sólo el cluster A depende exclusivamente de otro cluster B, y este último no es un cluster controlador (o cluster manejador) y además tiene un valor de IR por encima de $umbralIm$, entonces los elementos en B se agregan a A. Esta regla permite que un cluster (B) se una a otro cluster (A) cuando se da la situación particular de que el primero no satisface algunos requerimientos mínimos de importancia. En la Figura 7.2 **b** se puede visualizar un ejemplo donde se agrupan entidades de acuerdo a esta regla.

iii. Si dos clusters dependen mutuamente uno del otro, el cluster con menos valor de IR se agrupa en el que tiene el mayor valor, siempre que el primero no sea un cluster de controlador. Esta regla aplica un razonamiento que sigue el lineamiento planteado por las reglas i y ii. En la Figura 7.2 **c** se representa una situación donde esta regla permite agrupar dos clusters en uno.

Definir nuevos clusters

Este conjunto de reglas permiten la creación de nuevos clusters basados en nodos que todavía no han sido agrupados. Estas reglas poseen menor prioridad que aquellas que agregan nodos a clusters existentes. A continuación se exponen dos de las más relevantes:

i. Si un nodo posee un valor de IR mayor que $umbralIm$, entonces este nodo comprenderá un nuevo cluster;

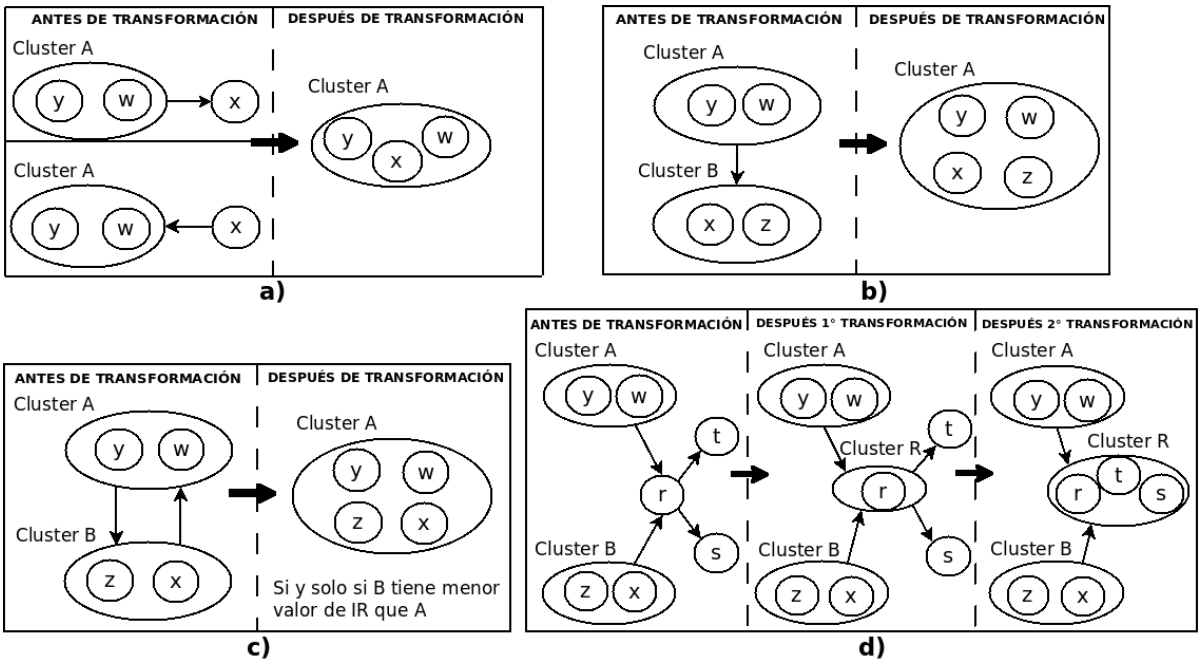

Figura 7.2: Representación de las reglas para agrupamiento de nodos. **a)** si un cluster depende de un nodo (o *vice versa*). En este caso, el cluster A y el nodo x. **b)** si un cluster A depende de otro cluster B. **c)** si dos clusters dependen mutuamente. En este caso, el cluster B posee menor valor de IR que el A. **d)** si dos o más clusters dependen de un nodo, entonces ese nodo comprende un nuevo cluster. En este caso , el nodo r conforma un nuevo cluster. Además, como segundo paso de transformación, se aplica la regla representada en *a)*.

ii. Si dos o más clusters dependen de un nodo, entonces el nodo constituye un nuevo cluster. En esta situación, el nodo es dependencia de más de un cluster y este no podría ser agrupado en alguno de los clusters. Por lo tanto, este nodo comprende un nuevo cluster rotulado con el mismo nombre que la signatura del nodo.

La Figura 7.2 **d)** representa un ejemplo donde se debe aplicar la regla ii para generar un nuevo cluster. En este caso en particular, el nodo r constituye un nuevo cluster. Luego los nodos t y s son agrupados en este nuevo cluster aplicando la regla representada en la Figura 7.2 **a)**.

El primer conjunto de reglas tiene una prioridad más alta que el segundo, que define nuevos clusters. Esto significa que SAPSI primero intenta agrupar los nodos en los clusters ya definidos antes de comenzar a definir los nuevos. El razonamiento detrás de estas heurísticas es dar más relevancia a los clusters que se han definido en etapas o ite-

raciones previas dentro de esta etapa. Generalmente, los clusters manejadores son más importantes que los clusters creados aplicando las reglas que definen nuevos.

Como se puede observar, el valor de $umbralIm$ se usa en dos situaciones particulares en donde permite resolver: i) cuándo un cluster puede "absorber" otro cluster, y ii) cuando un cluster nuevo puede ser generado a partir de un nodo no manejador. Sin el uso de este umbral, muchos clusters relevantes detectados serían "absorbidos" por otros y muchos nodos importantes se agruparían en otros clusters cuando en realidad deberían generar un nuevo cluster que sería relevante para el sistema. Un primer enfoque para determinar el valor del mismo es calculando el promedio de valores de IR para todos aquellos nodos que superaron el $umbralUt$, es decir, el umbral de utilidad definido para el filtrado. De esta manera sólo se tienen en cuenta aquellos nodos que pasaron el filtro y el valor de $umbralIm$ no se vería afectado por nodos irrelevantes.

Es importante destacar que todas las reglas en esta etapa se ejecutan hasta que no haya más modificaciones en la estructura. Finalmente, cada nodo que todavía no ha sido agrupado da lugar a un nuevo cluster; por ende, al final de esta etapa, cada nodo del GELMF está contenido en algún cluster.

7.2.4. Etapa 4: Estrategia de Rotulado de Clusters

Un factor importante en toda técnica de agrupamiento de Ingeniería Reversa es el nombramiento efectivo de los clusters. SAPSI rotula los clusters extrayendo términos de los componentes de la GUI. En etapas anteriores, la estrategia intenta nombrar automáticamente todos los clusters mediante el análisis de los términos relacionados con los componentes de la GUI del sistema. Sin embargo, en ciertas situaciones, estos términos no proporcionan un rótulo claro para el cluster. Por lo tanto, SAPSI proporciona al ingeniero de software información sobre los elementos del cluster y sus dependencias. De esta manera, este debe definir el rótulo más conveniente para el cluster en cuestión. Este aspecto es muy importante ya que un débil enfoque de nombramiento podría hacer que fracase toda la estrategia de comprensión (Tzerpos y Holt, 2000).

Rotulado Automático

La estrategia busca automáticamente los términos en el ARGUI con el objeto de encontrar nombres relativos al Dominio del Problema del sistema para ser usados en el rótulo del cluster. Para el caso de los clusters manejadores, el rótulo es determinado por el componente interactivo cuyo método/función manejador constituyó al cluster. Específicamente, el cluster es nombrado usando el rótulo contenido dentro del componente interactivo (por ejemplo, un botón). Si el widget no contiene un rótulo, el proceso de inspección continúa hacia los nodos descendientes en el ARGUI, hasta que se alcanza un componente con algún término. Por ejemplo, el nombre de la imagen utilizada para identificar el botón.

Aquellos clusters que no son manejadores son rotulados usando la signatura del método/función que es fuente dentro del cluster, es decir, aquellos nodos que no poseen arcos de entrada cuyos nodos origen pertenecen al mismo cluster.

Rotulado dirigido por el Ingeniero de Software

En determinados casos algunos clusters quedan sin rotular cuando la búsqueda automática no puede encontrar algún término para los mismos. Un claro ejemplo de esta situación es cuando un cluster no controlador contiene más de un nodo fuente. Además, pueden existir casos en donde la etiqueta asignada automáticamente para el cluster no sea adecuada o también sea posible asignar un término más acorde para el mismo. En tales casos, el ingeniero de software puede establecer un nombre para darle al cluster el rótulo más apropiado.

La información proporcionada al ingeniero de software está asociada con todos los datos recopilados durante la fase de extracción, específicamente se brinda la siguiente información:

1. Para el componente interactivo GUI asociado con el controlador que origina el cluster, se proveen los siguientes datos: rótulo del widget, identificador de la variable GUI en el ARGUI, tipo de componente y *tooltip*.

2. Cada signatura de los métodos/funciones dentro del cluster, destacando aquellos que tienen grado de entrada igual a cero con respecto a todo el sistema y aquellos

con grado de entrada cero con respecto al cluster. Esto permite distinguir aquellos nodos "iniciales" del cluster y del sistema.

3. Información relativa al cluster como el promedio de todos los valores de IR correspondientes a los métodos/funciones que integran el cluster, si es un cluster controlador y si es un cluster de inicialización (es decir, contiene los nodos de inicialización).

4. Relaciones *inter* e *intra* clusters.

El ingeniero de software puede establecer el rótulo para cada cluster analizando esta información, incluso en los casos en que el rótulo asignado automáticamente no fuera apropiado. Esta tarea debe ser llevada a cabo por el ingeniero de software ya que este posee el conocimiento y la capacidad de razonamiento para definir cada etiqueta teniendo en cuenta su experiencia y toda la información proporcionada.

7.2.5. Caracterización de la Técnica de Clustering Propuesta por la Estrategia

Una vez detallada la técnica de clustering propuesta por SAPSI, es posible destacar las características que la misma presenta en referencia a los conceptos y clasificaciones que se mencionaron al inicio de este capítulo en la Sección 7.1.

En primer lugar, es importante ubicar el enfoque dentro del contexto del método para generación de los clusters. En este sentido, el algoritmo presentado propone agrupar las entidades del GELMF de manera iterativa hasta llegar a un punto del agrupamiento donde no se puedan realizar más composiciones por medio de las reglas definidas. Por ende, el algoritmo es jerárquico aglomerativo, de acuerdo a la clasificación de Wiggerts (Wiggerts, 1997). En este sentido, la esencia del algoritmo puede ser comparada con ACDC (Tzerpos y Holt, 2000), ya que si bien no utiliza las mismas reglas, la base de ambos algoritmos es el uso de patrones que permitan extraer un modelo de cluster que ayude al ingeniero de software en la comprensión del sistema bajo estudio (Hamou-Lhadj y Lethbridge, 2004). Como se puede analizar, el algoritmo propuesto por SAPSI no define medida de similaridad alguna. La definición de la métrica IR es usada con propósitos

diferentes que una medida de similaridad. Como es posible determinar, la naturaleza del algoritmo no es maximizar una métrica de diseño sino obtener un modelo que refleje las funcionalidades del sistema y de esta manera asistir al ingeniero de software en la comprensión del mismo.

Por otra parte, también se define una estrategia de rotulado de clusters basada en la búsqueda de términos del Dominio del Problema. Esta posee una etapa automática donde principalmente se recorre el ARGUI analizando los componentes de la interfaz gráfica del sistema para encontrar rótulos de los clusters que están relacionados con los mismos. En el mismo sentido, también se usa la signatura de los métodos/funciones que componen cada cluster para rotular el mismo. Posteriormente, se plantea una etapa donde el ingeniero de software establece los rótulos de aquellos clusters para los cuales no ha sido posible encontrar términos de manera automática. Para esto la estrategia brinda información que caracteriza al cluster en cuestión.

Finalmente, como se verá en el Capítulo 9, se utilizan medidas que permiten comparar la descomposición generada por SAPSI con respecto a una descomposición generada por expertos. Esto sigue el criterio planteado por Shtern y Tzerpos, donde los autores afirman que una descomposición en el contexto de clustering de software, debería asistir al entendimiento del sistema antes que maximizar una métrica de evaluación. Es por este motivo que se usa la medida MoJo para comparar el modelo de clusters obtenido con uno autoritativo.

7.3. Construcción de Modelo de Casos de Uso UML

En esta sección se explica el procedimiento para transformar el modelo de cluster generado en las etapas previas a un modelo de caso de uso UML. El Algoritmo 3 presenta una descripción general de la fase de generación del modelo de caso de uso de UML. Este algoritmo toma como argumento el modelo de clusters representado en el GELMF y devuelve un archivo XMI (XML Metadata Interchange) con el modelo de casos de uso. Este tipo de archivos es un estándar *Object Management Group* (OMG) para la integración de diferentes tipos de herramientas y los mismos se pueden visualizar con cualquier herramienta de modelado UML que siga este estándar.

En primer lugar, el algoritmo inicializa dos variables auxiliares (*clusters* y *dependencias*) que contienen los artefactos principales del GELMF y crea el actor "usuario" en el modelo de caso de uso (líneas 1-4). Luego, cada cluster se mapea a un caso de uso del sistema (líneas 5-10), mientras que los arcos entre clusters se mapean a relaciones entre casos de uso (líneas 11-15). Dos o más dependencias (con el mismo sentido) entre clusters se asignan a una dependencia única entre los casos de uso correspondientes[3]. Además, cada caso de uso definido a partir de un cluster manejador está asociado con el actor (líneas 7-9), que en este caso representa al usuario del sistema.

```
   input  : grafo // GELMF con clusters.
   output: casoUsoXMI // modelo de casos de uso especificado en XMI.
   Data: clusters // conjunto de clusters que componen el GELMF.
   Data: dependencias // todas las dependencias entre clusters en el GELMF.
 1 inicializacion();
 2 clusters = grafo.obtenerClusters();
 3 dependencias = grafo.obtenerDependenciasEntreClusters();
 4 casoUsoXMI.agregarActor('Usuario');
 5 foreach cluster in clusters do
 6 |   cu = casoUsoXMI.agregarCasoUso(cluster.obtenerNombre());
 7 |   if cluster.esManejador() then
 8 |   |   casoUsoXMI.agregarAsociacionConActor('Usuario', uc);
 9 |   end
10 end
11 foreach dependencia in dependencias do
12 |   cuFuente =
   |     casoUsoXMI.encontrarCasoUso(dependencia.obtenerFuente());
13 |   cuDestino =
   |     casoUsoXMI.encontrarCasoUso(dependencia.obtenerDestino());
14 |   casoUsoXMI.agregarRealcionInclude(cuFuente, cuDestino);
15 end
```

Algoritmo 3: Algoritmo de construcción de modelo de caso de uso.

Todas las relaciones entre los casos de uso se definen como dependencias << *include* >>. La determinación de dependencias del tipo << *extend* >> y generalizaciones son planteadas como trabajos futuros. Este tipo de relaciones son menos frecuentes que las de << *include* >> y particularmente más complejas de detectar de manera automática (Si y cols., 2013).

[3]Si hay una dependencia entre dos nodos, entonces hay una dependencia entre los clusters que estos nodos integran.

En el Anexo B se muestra en detalle la aplicación del proceso de clustering a la aplicación libreta de contactos.

El capítulo 8 explica las características relevantes sobre la implementación de SAPSI y presenta la interfaz gráfica que permite al ingeniero de software definir rótulos para los clusters y analizar los aspectos pertinentes al modelo construido.

7.4. Notas y Comentarios del Capítulo

En este capítulo se presentaron los principales conceptos de clustering de software y las distintas técnicas utilizadas en el contexto de Ingeniería Reversa. Posteriormente se presentó la técnica de clustering utilizada por SAPSI para aglomerar y abstraer los componentes del GELMF. Dicha técnica incorpora conceptos de algoritmos de clustering diseñados en el marco de CP, los cuales hacen uso de detección de ciertos patrones orientados a obtener descomposiciones del sistema que ayuden al ingeniero de software a entender el mismo. Si bien se han propuesto diversas técnicas de clustering de software con el objetivo de extraer la arquitectura del sistema bajo estudio, esta tesis doctoral propone usar dichas técnicas para que el modelo de clusters obtenido permita construir un modelo de casos de uso. Para esto se basa en la detección de ciertos componentes de software que están fuertemente vinculados con las funcionalidades del sistema, como lo son los componentes interactivos de la GUI y aquellos métodos/funciones que implementan el comportamiento de dicho componente. Una vez generado el modelo de clusters, SAPSI ejecuta un algoritmo de transformación para construir un modelo de caso de uso a partir del modelo de clusters obtenido previamente.

Es importante destacar que el modelo de casos de uso obtenido por SAPSI puede ser distinto a aquel que fue construido en la etapa de diseño del sistema. Esto se debe a que el modelo obtenido por SAPSI está basado en el código fuente del sistema y el mismo ha sido construido durante la etapa de mantenimiento, mientras que el segundo fue generado en las etapas de gestación del sistema. Durante el desarrollo del sistema, específicamente entre las etapas previamente destacadas, es posible que la estructura del mismo cambie de tal forma que el modelo original represente sólo la base sobre la cual está desarrollado el código fuente. Específicamente, una de las etapas que más produce el comportamiento

antes descripto es la de mantenimiento y evolución. En términos generales, los modelos generados durante las etapas previas no son actualizados en concomitancia con el código fuente durante la etapa de mantenimiento.

Por otra parte, también es importante resaltar que una de las ventajas más valiosas de los casos de uso es que son fácilmente comprensibles para la mayoría de las partes interesadas involucradas en el desarrollo del sistema o procesos de reingeniería, es por esto que se presenta como un buen medio de comunicación. Además, es un modelo que está fuertemente ligado al Dominio del Problema, por lo tanto contiene mucha terminología referente al mismo. Este modelo brinda una perspectiva provechosa para el ingeniero de software proporcionando las principales funcionalidades del sistema y las dependencias entre las mismas. Por su parte, SAPSI permite identificar aquellos artefactos de software que implementan cada una de las funcionalidades visualizadas en el modelo de casos de uso.

Capítulo 8

Dupin: una Herramienta de Comprensión de Programas

La verdad no está siempre en el fondo de un pozo. En realidad, yo pienso que, en cuanto a lo que más importa conocer, es invariablemente superficial...

—Auguste Dupin

Este capítulo explica los conceptos y aspectos tecnológicos involucrados en la implementación de la estrategia desarrollada en los capítulos previos. Con el objetivo de mostrar los resultados obtenidos mediante la aplicación de la estrategia, se ha desarrollado *Dupin*[1], una herramienta que implementa todos los procesos explicados en los capítulos anteriores. Los módulos que componen el back-end del prototipo están desarrollados en Java y se utiliza la librería gráfica JUNG (JUNG-Team, 2016) para la visualización del GELMF (ver Figura 8.1).

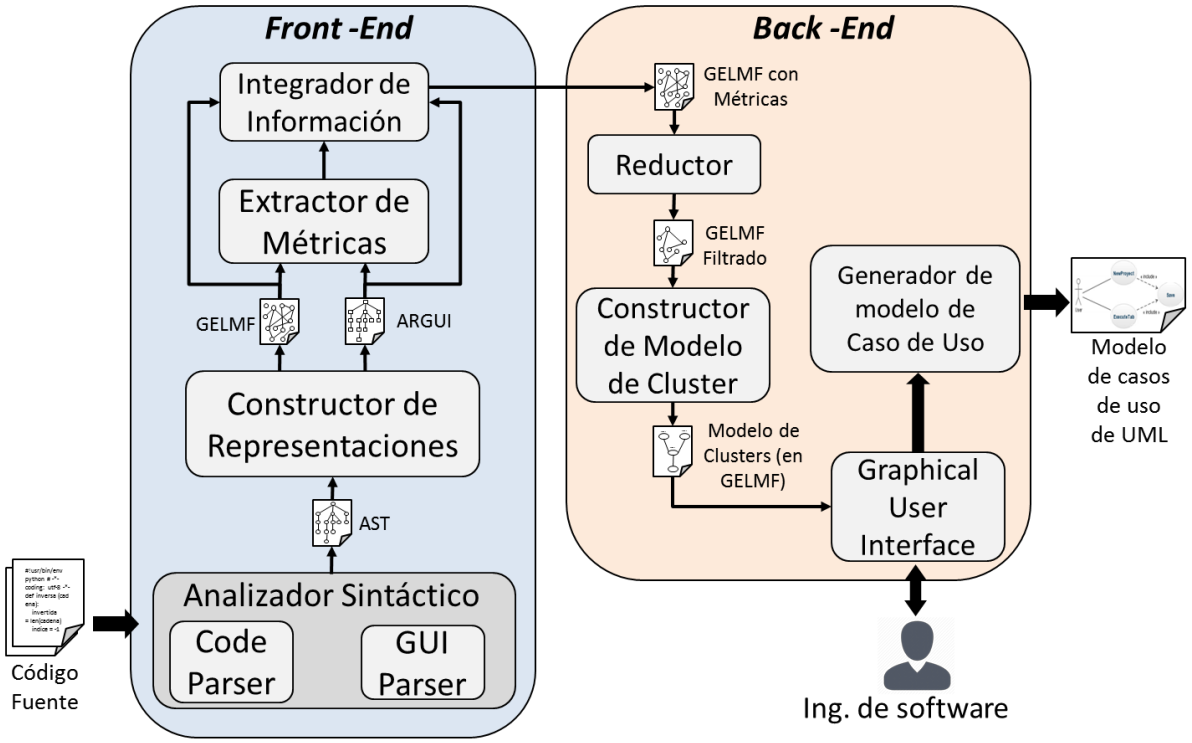

Figura 8.1: Arquitectura del prototipo Dupin.

8.1. Arquitectura de *Dupin*

En la Figura 8.1 se puede visualizar la arquitectura del prototipo desarrollado. A través de los módulos reflejados en la misma, es posible distinguir las distintas etapas de la estrategia. En un primer nivel, la arquitectura se encuentra dividida en dos grandes capas de módulos: por un lado los correspondientes al front-end y por el otro los que conforman el back-end. En los próximos apartados se detalla el funcionamiento de cada uno.

8.1.1. Módulos Correspondientes al Front-End

En el front-end de la aplicación se agrupan todos los módulos encargados de extraer la información y construir las representaciones de base que se utilizarán en las distintas etapas de la estrategia. Este conjunto de procesos varía de acuerdo al lenguaje analizado y las tecnologías asociadas al mismo. Es decir, si bien las técnicas principales y los

[1]Inspirado en *Aguste Dupin* un personaje del escritor Edgar Alan Poe, quien además fuera uno de los inspectores más conocidos de la literatura mundial.

procesos son los mismos (parsing, AST visitor, TEI estática), el front-end para sistemas desarrollados en lenguaje Java no será el mismo que para sistemas Python. Específicamente, es necesario contar con un *parser*, y un *visitor* específico para cada lenguaje. Incluso, los sistemas pueden estar desarrollados usando un mismo lenguaje pero diferentes librerías gráficas (por ejemplo, un sistema Python puede usar *GTK*, *QT*, *Tkinter*, o cualquier otra librería para implementar la parte de la GUI). Es por este motivo que es necesario contar con un analizador particular para cada librería gráfica a considerar, ya sea que el mismo esté embebido en el módulo *visitor* o bien, como se mencionó en en apartado 5.2.4, un analizador de los archivos que generan los diseñadores GUI.

Respecto al funcionamiento del front-end, en primer lugar recibe como entrada los artefactos del código fuente del sistema bajo estudio. Específicamente para esta versión del prototipo, la herramienta recibe el código fuente y el archivo generado por el editor de interfaces gráficas (por ejemplo, GLADE). Para el caso de este prototipo, se ha desarrollado un front-end que permite analizar sistemas desarrollados en Python y la librería gráfica GTK para desarrollo de GUIs.

El módulo *Extractor de Información*, haciendo uso de TEI estática, construye el AST a partir del código fuente del sistema. Dicho módulo está compuesto de un *parser* de código fuente y un *parser* GUI que se encargan de extraer la información de interés para la estrategia. El *Constructor de Representaciones* recibe el AST y en base al mismo construye las dos representaciones de base que necesita la estrategia para llevar a cabo el análisis, específicamente el GELMF y el ARGUI. Para la construcción del ARGUI toma como base el/los archivos generados por los diseñadores de GUI en conjunto con un parser XML. Además este módulo implementa el *visitor*, el cual se encarga de recorrer el AST en busca de toda la información requerida para la construcción de ambas representaciones, por ejemplo: llamadas a métodos/funciones, sentencias de conexión con widgets de la GUI (por ejemplo, establecer qué método/función implementa el comportamiento de cierto botón), determinación de los alcances de los métodos/funciones, entre otros.

Para el front-end de Python definido para esta investigación, la construcción del ARGUI se lleva a cabo usando el/los archivos generados por el diseñador GUI. Para esto se utiliza DOM (Domain Object Model), un parser para el lenguaje XML que construye explícitamente una representación interna del archivo analizado. Una vez generada

la representación interna, se le aplican distintos recorridos de árbol para recolectar la información de interés y finalmente poder construir el ARGUI.

Posteriormente, el módulo *Extractor de Métricas* recibe como entrada ambas representaciones y extrae el conjunto de métricas de software que utiliza SAPSI para atribuir el GELMF. Este módulo incorpora distintos algoritmos para extraer los distintos tipos de métricas. A continuación se especifica la forma en la que se obtiene cada métrica:

Peso GUI: se calcula usando la información provista por el GELMF en conjunto con el ARGUI. Es decir, se registra cada widget usado por cada método/función del GELMF. De esta manera es posible calcular el Peso GUI de acuerdo a la suma de los pesos asignados a los componentes GUI en función de la escala mostrada en el apartado 5.2.6.

Métricas de Código Fuente Estas métricas se calculan usando la información provista por el GELMF o también usando scripts python con algoritmos que faciliten la tarea. Existen numerosos aspectos que dependen del lenguaje con el cual esté desarrollado el sistema bajo estudio. Por ejemplo, para el front-end de Python, se utiliza el script *Lizard*[2], el cual es un analizador de código que permite calcular un conjunto de métricas básicas, entre estas, la cantidad de líneas de código y la complejidad ciclomática de McCabe de cada método/función. Por otra parte, de acuerdo a las construcciones dispuestas por el lenguaje, es posible determinar la *visibilidad* de cada método/función analizando la signatura del elemento (los nombres de los métodos privados son antecedidos por "__"). Para la determinación del *tipo* de nodo se debe analizar ciertas estructuras de datos construidas durante el proceso de creación del GELMF, específicamente aquellas que registran los alcances de los métodos/funciones. En este sentido, si la rutina se enmarca dentro de una clase, entonces es un método, en otro caso es una función.

Grado de Utilidad Estructural (SUD): esta métrica (Grado de Utilidad Estructural o por sus siglas en inglés *Structural Utility Degree* (SUD)) es estimada utilizando el algoritmo provisto por los autores (Hamou-Lhadj y Lethbridge, 2006). El algo-

[2] `http://www.lizard.ws`

ritmo toma como base los grados de entrada y salida de cada nodo en el GELMF, información fácilmente accesible a partir del uso de la librería de grafos JUNG.

Detalle de Implementación: esta métrica se obtiene analizando los rótulos de cada nodo del GELMF en busca de los patrones mencionados en el apartado 5.2.6.

Intermediación (Betweenness Centrality): el algoritmo para el cálculo de la intermediación de cada nodo es comúnmente conocido en el entorno de teoría de grafos. Para este caso en particular, se utiliza el algoritmo implementado por la librería JUNG que brinda los valores normalizados en el intervalo [0,1] (Brandes, 2001).

Finalmente, el módulo *Integrador de Información* atribuye el GELMF con toda la información extraída por los módulos predecesores, específicamente:

- todas las métricas de software mencionadas previamente (métricas indirectas);
- las métricas que se usaron para el cálculo de las anteriores (métricas directas), como por ejemplo el grado de entrada y de salida de cada nodo;
- la información que conforma la signatura de cada nodo, es decir, archivo, clase y nombre de método/función;
- colección de widgets del ARGUI que son usados por cada método/función, por ejemplo, rótulos, imágenes, paneles, menús, *tooltip*, etc;
- asociación de los métodos/funciones que implementan el comportamiento de cada widget.

Es importante remarcar que para admitir otros lenguajes de programación y librerías gráficas sólo se requiere cambiar los componentes principales del front-end. En pocas palabras, la estrategia dependerá de que el front-end provea las estructuras de base que son utilizadas para llevar a cabo todo el proceso de análisis. Para cambiar de lenguajes o reconocer distintos lenguajes, es necesario agregar los front-ends correspondientes para lograr la integración.

8.1.2. Módulos Correspondientes al Back-End

Una vez que se han generado las representaciones de base y que además se ha integrado toda la información extraída en el GELMF, la herramienta comienza con el conjunto de procesos que conforman el back-end. El GELMF atribuido es recibido por el módulo *Reductor* el cual implementa los procesos descriptos en el capítulo 6. Este módulo atribuye cada nodo del GELMF con la métrica IR, para que pueda ser utilizada en lo que resta del proceso. Además aplica un algoritmo de reducción iterativo sobre el GELMF eliminando aquellos nodos que no son relevantes para la lógica subyacente del sistema. El algoritmo requiere del establecimiento de un valor umbral que determine cuando un nodo debería ser considerado una *utilidad* en el contexto de la aplicación. Este valor debe ser establecido como parámetro de entrada antes del comienzo del análisis.

El GELMF filtrado es pasado como parámetro para el módulo *Constructor de Modelo de Cluster* que implementa todos los procesos descriptos en Sección 7.2. Una vez que todos los clusters han sido definidos de manera automática, el ingeniero de software debe analizar y proveer un rótulo para ciertos clusters. Específicamente, debe analizar aquellos clusters que no posean rótulo alguno o que posean uno que no es adecuado para el mismo, como se describe en la Sección 7.2. Además, la herramienta facilita la exploración del GELMF para determinar los siguientes aspectos del sistema:

- los métodos/funciones que componen cada cluster;
- las llamadas entre los métodos/funciones del GELMF;
- las relaciones entre los clusters en el GELMF;
- aquellos clusters estrechamente ligados con la GUI del sistema;
- los clusters que inicializan el sistema;
- los métodos/funciones que implementan cada funcionalidad del sistema;
- entre otros.

En esta instancia, es importante tener en cuenta que cada cluster representa un caso de uso del sistema bajo estudio.

Finalmente, una vez que el ingeniero ha establecido todos los rótulos necesarios, el módulo *Generador de Modelo de Caso de Uso* toma el GELMF con el modelo de clusters y lo transforma a un modelo de casos de uso de UML en formato XMI. Dicho proceso es explicado en la Sección 7.3.

Como se puede observar, el conjunto de procesos back-end llevan a cabo el análisis tomando en consideración muy pocos elementos relacionados con el lenguaje de programación. De esta manera se logra la independencia de la estrategia de aspectos tecnológicos, como por ejemplo, las construcciones sintácticas del lenguaje, las librerías gráficas utilizadas, entre otros.

8.2. Interfaz Gráfica de Usuario

Como se puede deducir, antes del último módulo que genera el modelo de casos de uso como resultado, el prototipo proporciona una GUI que permite al ingeniero de software establecer los rótulos y explorar el modelo de cluster para comprender mejor el sistema.

En la Figura 8.2 se puede visualizar una captura de la GUI de *Dupin* mediante la cual el ingeniero de software puede interactuar con la herramienta. En lo que resta de esta sección se describen los componentes que integran dicha GUI.

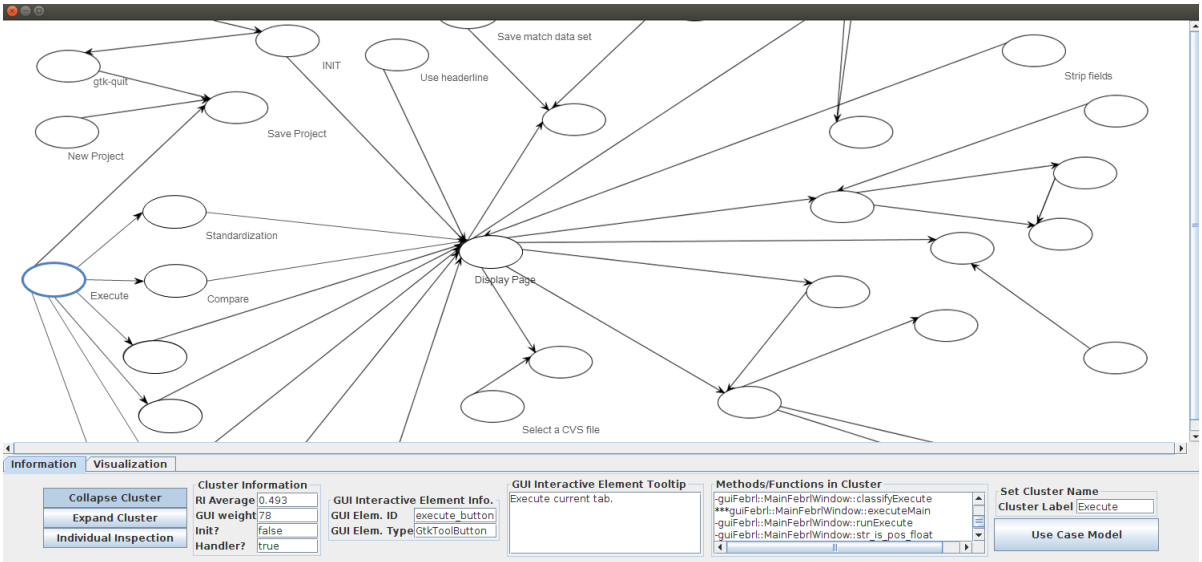

Figura 8.2: Captura de pantalla de la herramienta *Dupin* mientras el ingeniero de software define los rótulos para los clusters

8.2.1. Panel Central

El panel central visualiza el GELMF del sistema con todos los clusters definidos (nodos blancos con forma elíptica) y sus correspondientes dependencias (arcos dirigidos). El área de visualización abarca todo el ancho del prototipo ya que para sistemas grandes, el GELMF puede ser extenso. Además, por el mismo motivo la visualización cuenta con las siguientes funcionalidades de navegación:

- Arrastrar y Soltar: posibilita mover la estructura completa para hacer foco sobre una sección en particular de la misma.

- *Zoom-in* y *Zoom-out*: este tipo de funciones son ampliamente conocidas ya que exceden al ámbito de la informática. En este contexto, es importante contar con este de características básicas de navegación ya que la representación de un sistema de tamaño medio o grande puede llegar a ser difícil de visualizar. La funcionalidad de control de zoom se realiza mediante las teclas "+" y "-", y además se pueden ejecutar con la rueda del mouse.

- *Zoom to fit*: es un zoom especial ya que ajusta la vista de toda la representación usando todos los límites del área de visualización. Es importante para la exploración de un grafo ya que permite volver al estado inicial para visualizar toda la representación sin tener que hacerlo por medio de *zoom-in* y *zoom-out*. Esta funcionalidad está implementada con el botón de la rueda del mouse.

- Desplazamiento de elementos (nodos y clusters): esta funcionalidad permite seleccionar y mover los elementos del GELMF para desplazarlos hacia o desde un sector de visualización de interés.

- Selección de elementos (nodos y clusters): mediante la selección de algún elemento se puede visualizar información referente al mismo (ver subsección siguiente).

- Algoritmos de despliegue (*layout*): debido a que el GELMF se puede desplegar de diversas formas, se provee un conjunto de algoritmos de *layout* para facilitar la visualización del GELMF desde distintos puntos de vista. Dentro de estos, los

métodos dirigidos por fuerza (*force-directed*) presentan una buena elección, no solo por su amplio uso, sino por la naturaleza de su funcionamiento (Kobourov, 2012).

Todas las facilidades mencionadas previamente simplifican la tarea de exploración del GELMF por parte del ingeniero de software. Esta parte de la visualización fue implementada con la librería gráfica Java JUNG (JUNG-Team, 2016). Dicha librería es compatible con el lenguaje Java y provee un conjunto de características que facilitan el uso de la misma en este tipo de herramientas. Entre los aspectos más ponderables se pueden destacar los siguientes: presenta un gran conjunto de funciones de navegación integradas, escala bien en grafos de tamaño medio o grandes, soporta una amplia gama de representaciones de grafos, incorpora implementaciones eficientes de algoritmos de teoría de grafos, provee varios ejemplos donde muestra el uso de las distintas funcionalidades, entre otras.

8.2.2. Panel Inferior

El panel inferior muestra un conjunto de campos de texto que proporcionan información sobre el elemento seleccionado. A continuación se describe el propósito de cada uno de estos:

Información inherente al Cluster: muestran información de ciertas métricas del cluster seleccionado, específicamente, el promedio de valores IR en todo el cluster, la suma de los pesos GUI, si es un cluster inicializador y si es un cluster manejador. Por otra parte, si un nodo es seleccionado, estos campos muestran la misma información, sólo que específica para dicho nodo.

Información del Componente GUI Interactivo: Si un cluster fue creado a partir de un nodo manejador de algún componente GUI interactivo, tales como botones, items de menú, etc. estos campos muestran información referente a dicho componente. Más precisamente, el *tooltip*[3], el nombre y el tipo de este widget en el ARGUI (por ejemplo, para un botón en el caso de la librería Swing, indicaría *JButton*).

[3]información emergente

Componentes del Cluster: Estos campos muestran la signatura de todos los métodos/funciones que están agrupados en el cluster seleccionado. Además, para cada método/función en el cluster, el área de texto señala si se trata de un nodo "fuente" dentro del cluster (esto se señaliza con "***" al comienzo de la signatura del elemento). Este aspecto resulta importante porque los nombre de los nodos origen podrían ayudar significativamente a definir el rótulo para el cluster seleccionado.

Botones de navegación de cluster: este grupo de botones facilita la inspección de cada cluster en particular, a continuación se indica la funcionalidad de cada uno:

- *Collapse Cluster:* agrupa visualmente todos los nodos que componen cada cluster visualizando un único elemento que representa el mismo. Si no están colapsados, los nodos que pertenecen a un mismo cluster son visualizados con un mismo color.
- *Expand Cluster:* es la operación inversa a la anterior, es decir expande el cluster seleccionado para poder visualizar los nodos y relaciones internas que componen al mismo.
- *Individual Inspection:* el cluster seleccionado es analizado de manera independiente visualizando de manera aislada sólo aquellos nodos y relaciones que pertenecen al mismo.

En conjunto, estos botones permiten al ingeniero de software navegar el GELMF, analizar la composición de los clusters y estudiar las relaciones internas/externas entre los nodos. Claramente, estas características mejoran el proceso de inspección del sistema en estudio.

***Set Cluster Name*:** por medio de este campo de texto se permite definir el rótulo de cada cluster tomando como base toda la información provista referente al mismo.

Botón "Use Case Model": finalmente, una vez que el ingeniero de software ha completado la fase de rotulado, hace clic en el botón "Use Case Model" del panel inferior para generar el modelo de caso de uso para el sistema bajo estudio. Esto se explica en detalle en el apartado 7.3. Este modelo se genera en un archivo XMI y se

puede visualizar en cualquier herramienta de modelado UML que siga el estándar OMG (por ejemplo, *Modelio* o *Bizagui*), lo que permite la visualización del modelo de caso de uso obtenido. Los métodos/funciones implementan cada caso de uso son incorporados al XMI usando el *tag* `comment`[4].

Como es posible analizar, la GUI del prototipo muestra distintos aspectos del cluster para asistir al ingeniero de software a inspeccionar dicho modelo y además a rotular de manera conveniente los clusters que lo integran.

8.3. Notas y Comentarios del Capítulo

En este capitulo se presentaron los aspectos relativos a la implementación de la estrategia SAPSI desarrollada en los capítulos previos. Para esto se desarrolló *Dupin* una herramienta que implementa todos los procesos y actividades propuestos por SAPSI. La arquitectura del prototipo presenta una distribución de módulos agrupados en dos grandes grupos: *front-end* y *back-end*. El primero agrupa aquellos módulos que extraen información del sistema bajo estudio y construyen la representaciones de base que serán usadas durante todo el proceso de análisis. El mismo está fuertemente relacionado con las características relativas a la plataforma y las tecnologías usadas por el sistema, es decir, los aspectos cercanos a las construcciones sintácticas del lenguaje de programación, las librerías gráficas empleadas, otras características del lenguaje, entre otros. El grupo de módulos back-end es independiente del lenguaje utilizado e implementa la parte esencial de la estrategia. Este tipo de arquitecturas de herramientas de CP permiten aplicar la estrategia a distintos lenguajes de programación y de esta forma abarcar una gama más amplia de sistemas. De esta manera, el reconocimiento de otros lenguajes de programación y librerías gráficas se reduce al desarrollo de los front-ends para los mismos.

Para el caso de este prototipo, se ha desarrollado un front-end que permite analizar sistemas desarrollados en python y la librería gráfica GTK para desarrollo de GUIs. La construcción del GELMF se realiza mediante el uso de técnicas de parsing mientras que

[4]Si bien dicho tag no ha sido especificado para tal fin, es un elemento del XMI que menos afecta la semántica del modelo.

para la construcción del ARGUI se toma como base el/los archivos generados por los diseñadores de GUI en conjunto con un parser XML.

La interfaz gráfica del prototipo ofrece distintas opciones de navegación para explorar el modelo de cluster, desde facilidades como funciones de *zoom* hasta información de la composición y las relaciones de cada cluster. Por otra parte, también es posible establecer el rótulo de aquellos clusters que no posean uno o aquellos para los cuales se obtenga uno más adecuado.

Es importante resaltar que la estrategia toma como base sólo aquellos artefactos que pueden ser extraídos a partir del código fuente para llevar a cabo el análisis. Dichos artefactos son: los métodos/funciones de todo el proyecto, las dependencias estáticas entre los mismos, la estructura de la GUI y sus componentes y las métricas de software. En este sentido, en concordancia con varios autores de la disciplina, este es el recurso más confiable y en la mayoría de los casos, es el único con el que se cuenta a la hora del análisis. Esto hace a la estrategia independiente de otro tipo de artefactos de gestión ajenos al código fuente como por ejemplo información relativa a los equipos de desarrollo, modelos de las etapas de construcción del sistema, representaciones del Dominio del Problema, entre otros. Si bien este tipo de información es valiosa, en la mayoría de los casos es difícil de reconstruir/obtener.

Capítulo 9

Evaluación de la Propuesta

Dentro de cada programa extenso, hay un pequeño programa tratando de salir.

— Charles Antony Richard Hoare

En los capítulos precedentes se presentó una estrategia para asistir al ingeniero de software durante el proceso de Comprensión de Programas. La misma plantea vincular artefactos del Dominio del Programa con ciertos componentes del Dominio del Problema, de manera tal de que el ingeniero de software pueda analizar las funcionalidades que posee el sistema, pudiendo inspeccionar las mismas en distintos grados de abstracción. Con este objetivo, se extrajo un modelo de casos de uso de UML para el sistema bajo análisis, el mismo sirve como base para visualizar e inspeccionar esta vinculación. Como se ha explicado en capítulos anteriores, este modelo posee ciertas ventajas que lo hacen atractivo con este tipo de propósitos. Para la extracción de este modelo, se desarrolló una herramienta que implementa todos los procesos y conceptos descriptos a lo largo de este informe.

En este capítulo se presentan todos los aspectos que posibilitan la evaluación de la propuesta. Específicamente, en la Sección 9.1 se explican los lineamientos tenidos en cuenta para poder evaluar la estrategia, mientras que en la Sección 9.2 se exponen los resultados de aplicar la estrategia propuesta a dos sistemas escritos en lenguaje Python.

9.1. Metodología de Evaluación

Debido a la naturaleza de la propuesta, es difícil evaluar los resultados de manera sistemática y precisa. En este sentido, es necesario tener en cuenta los siguientes factores a la hora de proponer una metodología de evaluación:

- El modelo de casos de uso obtenido por SAPSI no debería ser comparado con uno generado en las fases de diseño del sistema [1]. Esto se debe a que el modelo creado en las primeras etapas del desarrollo del sistema no contiene la misma precisión que el modelo que genera SAPSI (Shtern y Tzerpos, 2011). Por otra parte, ciertos factores hacen que el sistema vaya cambiando (a veces de manera substancial) con respecto al modelo de casos de uso inicial, como por ejemplo las etapas de implementación y mantenimiento; el proceso de erosión del software (De Silva y Balasubramaniam, 2012), entre otros. Por lo general, los modelos iniciales no son actualizados a lo largo de las etapas subsiguientes (Smit y cols., 2008; Bojic y Velasevic, 2000).

- Una de las ventajas de usar el modelo de casos de uso a la hora de realizar tareas de CP, se vuelve un obstáculo a la hora de evaluar la propuesta. Como bien se remarcó en distintos capítulos, el modelo de casos de uso es considerado un buen medio de comunicación, ya que es fácil de entender por los involucrados en el desarrollo del sistema. Además, está fuertemente ligado al Dominio del Problema más que el Dominio del Programa, es decir, es un modelo que ofrece un alto grado de abstracción con respecto al código fuente del sistema. Sin embargo, esto hace que la medición de la precisión de los resultados sea compleja y requiera de un análisis minucioso de los modelos obtenidos para responder a ciertas dudas que pueden surgir como por ejemplo si un caso de uso determinado fue correctamente extraído, o si los métodos/funciones que lo componen están correctamente asignados.

De los factores expuestos previamente, es posible determinar las siguientes consideraciones:

- Para ponderar los resultados obtenidos, es necesario generar un modelo ideal (o "gold standard"), el cual debe ser construido por ingenieros de software para garan-

[1] Además, es difícil conseguir estos modelos y en caso de estar disponibles es poco frecuente que se encuentren actualizados.

tizar una mayor precisión en la composición del mismo. Este servirá como "ideal" al cual se debe asemejar el modelo generado por la estrategia de CP.

- La evaluación de este tipo de propuestas requiere la asistencia de ingenieros de software que permitan esclarecer determinadas situaciones que pueden presentarse a la hora de evaluar esta clase de modelos. Por ejemplo, si dos casos de uso representan lo mismo, aunque no estén compuestos por exactamente los mismos elementos o estén relacionados de manera diferente.

En esta sección se desarrolla una metodología para evaluar los resultados obtenidos por SAPSI. Para esto, en primer lugar se presenta un procedimiento para la generación de un modelo de casos de uso ideal para el sistema. Posteriormente, se presentan dos preguntas de investigación que se tomarán como principales aristas para conducir la evaluación de los modelos obtenidos. Finalmente, se presenta el método de comparación de modelos propuesto que facilita contrastar el modelo obtenido con SAPSI con un modelo "autoritativo" generado por ingenieros de software.

9.1.1. Procedimiento para Generar Modelo Ideal

En el contexto de Ingeniería Reversa, es muy común que los sistemas inspeccionados no presenten los correspondientes modelos ideales para el propósito del análisis. Los casos de estudio analizados en esta investigación no son la excepción. Para lograr este objetivo, se presenta un procedimiento que facilita la obtención de un modelo de casos de uso ideal con fines de comparación. Dicho procedimiento cuenta con los siguientes pasos:

1. Distribuir los miembros del Grupo de Ingeniería de Software en subgrupos
2. Obtener de un Modelo Básico por cada Subgrupo
3. Construir un Modelo Completo por cada Subgrupo
4. Definir un único Modelo Final entre todos los miembros

En los siguientes apartados se explica más detalladamente los pasos mencionados describiendo las tareas más relevantes que son llevadas a cabo por cada uno de estos.

Distribución del Grupo de Ingeniería de Software

En primer lugar, se solicita a un grupo de ingenieros de software realizar tareas de Ingeniería Reversa al sistema bajo estudio con el objetivo de obtener un modelo ideal que posibilite comparar los resultados obtenidos con la herramienta *Dupin*.

A fin de promover la imparcialidad en la obtención del modelo ideal, los miembros del grupo de ingeniería deben ser distribuidos en subgrupos pequeños conformados por 2 o 3 integrantes.

Obtención de un Modelo Básico por cada Subgrupo

Cada subgrupo de ingenieros debe obtener un modelo de caso de uso ideal para el sistema analizado. Considerando que no es directo llevar a cabo este tipo de tareas de manera manual exclusivamente inspeccionando el código, mucho más en sistemas de software de tamaño medio o grande, dicho modelo puede ser construido utilizando los siguientes recursos:

- Documentación disponible del sistema bajo estudio
- Foros, wikis y cualquier información informal del sistema
- Técnicas y/o Herramientas de Ingeniería Reversa

Con motivos de reducir el tiempo que puede demandar la construcción de este tipo de modelos, se recomienda utilizar la técnica *Análisis Comportamental* (Berón, 2010) la cual propone la captura de información a partir de la reproducción de cada comportamiento del sistema. La documentación del mismo sirve como soporte básico para la determinación de los diferentes escenarios que abarquen las funcionalidades esenciales reveladas en la GUI del sistema. Luego, se realizan ejecuciones de estos escenarios y de esta manera se construye un primer modelo de caso de uso básico[2].

Construcción del Modelo Completo por cada Subgrupo

Con el objetivo de que cada subgrupo produzca un modelo comparable con el obtenido por SAPSI, es necesario que el modelo básico obtenido en el paso anterior sea extendido

[2]El mismo se considera básico ya que sólo releva las funcionalidades perceptibles a través de la GUI.

considerando los siguientes aspectos:

- *Casos de Uso internos*: el modelo básico posee sólo las funcionalidades "perceptibles" a través de la GUI, si bien estas son las de mayor interés para el análisis, un modelo completo puede contener casos de uso que representen funcionalidades relevantes invocadas de manera interna, es decir, que son invocadas por otros casos de uso y no de manera directa a partir de la interacción del usuario con el sistema.

- *Artefactos del Dominio del Programa en el Modelo:* el modelo obtenido por SAPSI ofrece la posibilidad de consultar los métodos/funciones que implementan cada caso de uso del sistema. Por ende, los ingenieros deben determinar qué artefactos del código fuente han sido utilizados para implementar cada caso de uso del modelo ideal.

Teniendo en cuenta lo antes descripto, los ingenieros de software son provistos con un script que implementa un *tracer* específico de acuerdo al front-end utilizado por la estrategia. Claramente, el mismo dependerá del tipo de sistema analizado; para el caso de los inspeccionados en esta tesis se proporcionó un *tracer* para Python. Por medio de esta herramienta es posible ejecutar los distintos escenarios y obtener todos los métodos/funciones asociados con cada caso de uso extraído. Este script ayuda al ingeniero a registrar las relaciones llamador/llamado y a listar los métodos o funciones que han sido utilizados durante la ejecución de un escenario determinado. Con el objetivo de garantizar que la técnica de clustering de SAPSI y los ingenieros de software puedan construir sus modelos utilizando el mismo conjunto básico de métodos/funciones, el *tracer* provisto también debe incorporar el proceso de reducción de información de SAPSI[3].

En el caso específico del front-end para Python, el *tracer* fue implementado usando un enfoque similar al propuesto por Hamou-Lladj (Hamou-Lhadj y Lethbridge, 2006), quien desarrolló una herramienta de este tipo que filtra métodos basándose en un grafo de llamadas a funciones estático. Además, el código fuente del sistema y el GELMF sirvieron como soportes secundarios para complementar el análisis en algunas situaciones particulares (por ejemplo, para comprender ejecuciones de sentencias de iteración

[3]Como se verá más adelante en el capítulo, este aspecto también es condición necesaria para el uso de la métrica de comparación entre modelos de clusters *MojoFM*

específicas o para detectar cambios de flujo de ejecución por declaraciones de selección).

Después de la ejecución de los escenarios utilizando el *tracer* y los soportes secundarios, los ingenieros de software llevan a cabo tareas de reestructuración para obtener un modelo más específico, además se identifican los métodos/funciones relacionados con cada caso de uso extraído. Estas tareas implicaban unir y/o dividir casos de uso, agregar y eliminar relaciones y asignar métodos/funciones a cada caso de uso en el modelo.

Definición del Modelo Final

Una vez que cada subgrupo obtiene un modelo de casos de uso completo para el sistema bajo estudio, es necesario unificar todos los resultados en un único modelo ideal. Con este objetivo se llevan a cabo una o más sesiones de trabajo donde se coordina y convienen las diferencias entre los modelos finales de cada subgrupo para de esta forma lograr obtener el modelo de casos de uso ideal.

9.1.2. Preguntas de Investigación

Se establecen las siguientes preguntas que permitirán evaluar la precisión del modelo obtenido con SAPSI:

- ***RQ1:*** ¿Cuán exacto es el modelo de caso de uso extraído por SAPSI con respecto al modelo ideal? Esta pregunta tiene como objetivo la estimación de la precisión del enfoque, medida a nivel de detección de caso de uso.

- ***RQ2:***

  ¿Cuán precisa es la detección de métodos/funciones dentro de cada cluster cuando se compara con el modelo ideal? Esta pregunta tiene como objetivo estimar la precisión del enfoque de clustering, medido en términos de la composición de cada cluster obtenido en el modelo.

9.1.3. Comparación de Modelos

Comparar los modelos obtenidos con los ideales es un aspecto esencial cuando se obtienen resultados de manera automática. De por sí, no existe una única manera de

hacerlo y cualquier método trae aparejado determinados aspectos que lo hacen dependiente de factores externos al mismo. Es decir, el enfoque de comparación puede estar influenciado por una serie características inherentes al tipo de modelo comprado. Por ejemplo, se puede comparar dos casos de uso teniendo en cuenta los rótulos o de acuerdo a cómo están relacionados con los demás casos de uso y los tipos de relaciones. En este contexto, a continuación se presentan dos medidas que intentan responder a las preguntas de investigación.

Método de comparación de Modelos de Caso de Uso (*RQ1*)

El método de comparación adoptado para este tipo de artefactos propone evaluar el modelo de caso de uso obtenido tomando como punto de comparación los elementos del diagrama de caso de uso de UML. La comparación tiene como objetivo evaluar la precisión de la estrategia para obtener un modelo de caso de uso significativo del sistema bajo análisis.

Es importante destacar que, de la bibliografía analizada en el contexto de esta tesis doctoral, no se encontró una medida de similaridad que compare dos modelos de caso de uso de UML. En contraste con esto, los modelos de clusters han sido analizando con más profundidad en este sentido. Varios autores han propuesto distintas medidas de similaridad a través de los años, como por ejemplo las familia de métricas *MoJo* (Tzerpos y Holt, 1999; Wen y Tzerpos, 2004) o *KE* (Koschke y Eisenbarth, 2000).

Es por este motivo que para evaluar ambos modelos (el obtenido y el ideal) se toma en consideración la proporción de coincidencias teniendo en cuenta cada tipo de elemento del diagrama de casos de uso que presenta UML. A continuación se mencionan los elementos que se consideran para la comparación:

- *Actores*;
- *Casos de Uso*;
- *Relaciones* que a su vez se clasifican en *Include*, *Extend*, *Generalización*, *Asociación* (actor-caso de uso).

Debido a que la estrategia no examina relaciones del tipo *extend* y *generalizaciones*, dichos elementos del diagrama se excluyen de la comparación.

Para cada tipo de componente en el diagrama, se analizan las coincidencias sobre el conjunto total de elementos inspeccionados. Específicamente, si `B` representa los componentes del modelo ideal y `A` los componentes del modelo producido por SAPSI, se calcula la proporción de coincidencias entre ambos modelos sobre el total de elementos. Esta actividad se lleva a cabo para cada tipo de elemento en el diagrama. La misma se expresa en la Ecuación 9.1.

$$Com(A, B) = \frac{|A \cap B|}{|A \cup B|} * 100\,\% \tag{9.1}$$

Los elementos más relevantes a ser comprados son los casos de uso. Sin embargo, también son los más complejos de analizar ya que no existe un criterio preciso y consistente para definir cuándo dos casos de uso representan lo mismo. En función de esto, se establece que un caso de uso de `A` es análogo a un caso de uso en `B` cuando hay un conjunto de similitudes entre ellos, tales como etiquetas, componentes (métodos/funciones) y relaciones con otros usos casos. Como se puede deducir, esta comparación se debe llevar a cabo con el asesoramiento de expertos en Ingeniería de Software que además hayan analizado el sistema bajo estudio para lograr consenso respecto de la correspondencia entre los casos de uso. De no ser así, es decir, si se comparan de manera sistemática, la precisión de la propuesta puede verse afectada drásticamente, ya que puede que las coincidencias entre modelos no sean idénticas.

Luego de la comparación de los casos de uso, todos los tipos de relaciones restantes son comparados. Para este caso, se toma en cuenta un enfoque básico: dos relaciones, una en `A` y otra en `B`, representan la misma si conectan el mismo par de casos de uso en ambos modelos y además ambas relaciones son del mismo tipo.

Medida de Comparación para Modelos de Cluster (*RQ2*)

Esta medida intenta calificar la asociación de métodos/funciones a los clusters (casos de uso). Para esto, se utiliza una métrica de similaridad denominada *MoJo*, la misma es reconocida y ha sido ampliamente utilizada en el contexto de clustering (Tzerpos y Holt, 1999). En pocas palabras, MoJo intenta capturar la distancia entre dos descomposiciones como el número mínimo de operaciones *Move* y *Join* que deben realizarse para

transformar una agregación en la otra.

Las operaciones *Move* implican la reubicación de una entidad de un cluster a otro (o a un nuevo cluster), mientras que una operación *Join* toma dos clusters y los fusiona. Como se puede interpretar de la lectura de los trabajos de la propuesta original, esta medida no es simétrica. Por este motivo, los autores también introdujeron la medida de eficacia *MoJoFM* (Ecuación 9.2) que se basa en la distancia *MoJo* pero asigna números en el intervalo [0,100] y que además es independiente del tamaño de las descomposiciones (Wen y Tzerpos, 2004):

$$MoJoFM(M) = (1 - \frac{mno(A,B)}{max(mno(\forall A,B))}) * 100\,\% \tag{9.2}$$

donde `M` es la técnica de clustering de software examinada, `A` es la descomposición creada automáticamente, y `B` es la descomposición ideal (o como se referencia en los trabajos de clustering, "autoritativa"). Además, `mno(A,B)` es el número mínimo de movimientos (*move*) y operaciones de unión (*join*) que deben realizarse para transformar `A` en `B`. Wen y Tzerpos (Wen y Tzerpos, 2004) explican específicamente esta métrica de similaridad y las mejoras incorporadas en contraste con la *MoJo* original.

Es importante resaltar que la familia de métricas *MoJo* requiere que el conjunto de métodos/funciones que posee cada agrupamiento a comparar sea el mismo. En esta investigación en particular, se ha examinado el conjunto de métodos y funciones asignados en el modelo ideal por los ingenieros de software. El criterio detrás de este razonamiento es que dicho modelo siempre contendrá menos artefactos ya que se desarrolla utilizando información dinámica, la cual permite inspeccionar los artefactos utilizados durante la ejecución de un escenario en particular, pero no comprende todos los escenarios posibles.

9.2. Resultados de la Evaluación

Con el objetivo de evaluar la contribución de SAPSI, esta sección presenta el análisis de dos casos de estudio:

- *Freely Extensible Biomedical Record Linkage (FEBRL):* software de código abierto que implementa diversas técnicas avanzadas desarrolladas para limpieza y estan-

darización de datos, indexación (blocking), comparación de campos y clasificación de registros por pares; además la herramienta encapsula todas las funcionalidades en una aplicación con interfaz gráfica de usuario (Christen, 2008). El proyecto completo tiene aproximadamente 60 KLOCs[4] y usa la biblioteca PyGTK y Glade (Galde, 2016) para el diseño de una GUI independiente de la plataforma.

- *Pigeon Planner:* software multiplataforma de código abierto para control, organización y gestión de palomas. Con una interfaz gráfica que permite al usuario introducir todos los detalles de cada paloma, registrando entre otras características, el pedigrí, detalles de color y datos del linaje, además del número de anilla correspondiente y fecha de nacimiento. Asimismo dispone de varias opciones para ordenar las líneas de reproducción por edad, sexo o comportamiento en vuelo; se pueden agregar imágenes a la ficha de cada ave así como la distancia o el clima en las carreras en las que haya participado. Además provee un conjunto de herramientas tales como una calculadora de velocidad, un gestor de copias de seguridad y un panel de estadísticas generales. El proyecto completo tiene aproximadamente 32 KLOCs y usa la librería gráfica GTK y Glade (Galde, 2016) para el diseño de una GUI independiente de la plataforma.

Es necesario resaltar en referencia al tamaño de los sistemas analizados, que el lenguaje de programación Python posee ciertas características que hacen a la longitud de los mismos de 3 a 5 veces más corta que otros lenguajes de propósito general como $C\#$, $C++$ o $Java$ (Prechelt, 2000).

Esta sección se organiza de la siguiente manera: en la subsección 9.2.1 se describen los aspectos relativos la configuración de la estrategia con la cual se llevaron a cabo los experimentos, mientras que en las subsecciones 9.2.2 y 9.2.3 se presentan los resultados obtenidos a partir del análisis de los sistemas Febrl y Pigeon Planner correspondientemente.

[4]Esta medida indica cantidad de líneas de código sin tener en cuenta espacios en blanco, por lo tanto 1KLOC equivale a 1000 líneas de código del lenguaje analizado

9.2.1. Configuración de SAPSI

Como se describió en capítulos precedentes, la estrategia requiere la determinación de ciertos valores que establecen la configuración inicial para llevar a cabo la inspección de los sistemas. Dichos valores deben ser estimados tomando como base las siguientes consideraciones:

- las características que presenta el sistema bajo análisis, como por ejemplo lenguaje/s de programación, estimaciones del tamaño del proyecto (número de archivos o $\#LOCs$), las herramientas externas empleadas y/o paradigmas de programación predominantes;
- Experiencia con la estrategia SAPSI.
- Análisis de los resultados usando diferentes configuraciones para el mismo sistema

En los siguientes aparatados se define la configuración de la estrategia utilizada para analizar los casos de estudio presentados en esta sección.

Valor de Umbral de Utilidad (`umbralUt`)

En primer lugar, el proceso de reducción intenta eliminar detalles de implementación de GELMF; especialmente aquellos nodos que no representan métodos o funciones esenciales con respecto a la lógica subyacente del sistema. Antes del comienzo del proceso de filtrado, se debe definir un valor que represente el *umbral de utilidad* (`umbralUt`), el mismo debe estar contenido en el intervalo [0,1]. Este parámetro delimita cuando un método/función se reconoce como un detalle de implementación. De esta manera, todos los métodos/funciones cuyos valores *IR* están por encima de ese umbral se eliminarán del GELMF.

Usualmente, el *umbral de utilidad* es establecido por expertos de dominio, dependiendo del sistema examinado. Para este estudio, se realizaron diversas ejecuciones de los sistemas estableciendo diferentes valores para este umbral. En particular, para los dos sistemas analizados se cumple que para valores de `umbralUt` en el intervalo [0.05,0.25], el número resultante de elementos filtrados no difiere en gran medida de una ejecución a otra. Sin embargo, la cantidad de elementos filtrados aumenta cuando se definen valores

más altos. Para el caso de estudio presentado en este trabajo, se definió un valor de umbral relativamente bajo con el fin de asegurar que el proceso de agrupamiento tome como base los elementos esenciales para lograr la agregación. Para los casos de estudio desarrollados, se estableció un valor de umbral de 15 % (`umbralUt = 0.15`). Esto permite que la fase de filtrado elimine solo aquellos nodos cuyo valor *IR* es relativamente bajo.

Debido al tamaño de los sistemas se optó por el enfoque iterativo de reducción de información en ambos casos con el objetivo de obtener un filtrado que resulte en un GELMF con aquellos métodos/funciones más influyentes para cada sistema.

Es importante remarcar que existe un gran conjunto de nodos y arcos filtrados que se eliminan por dos razones esenciales: i) numerosos detalles de implementación obtienen sistemáticamente $IR = 0$ y estos serán eliminados sin importar el valor umbral, y ii) en cada iteración del proceso de filtrado, muchos nodos quedan aislados del resto y son eliminados en iteraciones posteriores.

Valor de Umbral de Importancia Relativa (`umbralIm`)

Otro umbral que debe establecerse al comienzo del proceso es el valor de `umbralIm`, como se especificó en la Subsección 7.1.2. El mismo establece cuándo un método/función o cluster puede influir en la lógica subyacente del sistema con respecto a la comprensión del mismo. En este caso de estudio, se estableció `umbralIm` como el promedio de los valores de *IR* para todos aquellos nodos que superaron el *umbralUt*, es decir, el umbral de utilidad que se definió para el filtrado. De esta manera sólo se tienen en cuenta aquellos nodos que superaron el filtro y el valor de *umbralIm* no es afectado por valores bajos de IR.

9.2.2. Sistema FEBRL

En este apartado se examinan los distintos aspectos que resultaron de inspeccionar el sistema Febrl utilizando la estrategia SAPSI. En primer lugar se analizan los resultados obtenidos en la etapa de extracción, representación y filtrado de la información.

La Figura 9.1 muestra la Estructura de Agregación de SAPSI con todas las preferen-

cias para la obtención de IR para el método "guiFebrl.MainFebrlWindow.writeStatusBar". Incluso cuando "writeStatusBar" obtuvo valores aceptables para *Métricas de código fuente* y *Peso GUI*, como resultado de los valores bajos obtenidos para *Intermediación* y *SUD*, el método se consideró como una *utilidad*. Los operadores *C+-* y *C-+* son mandatorios, por lo tanto, propagan el 0 (cero) a la Preferencia Global, que en este caso representa el valor de *IR*.

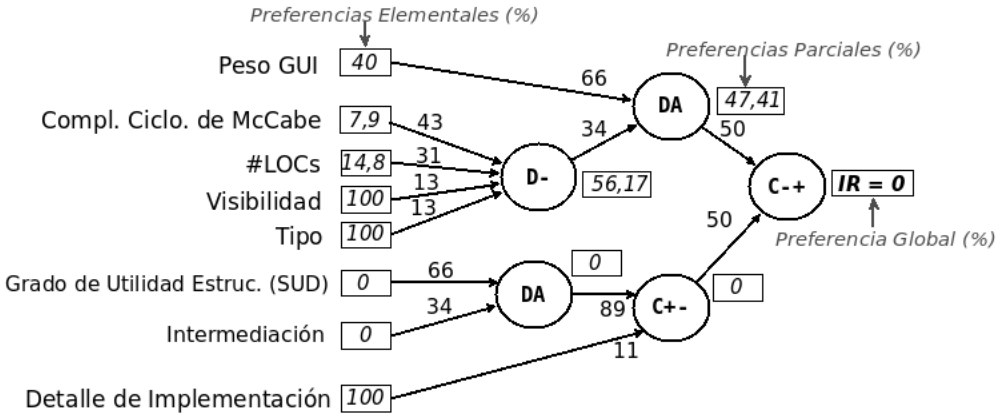

Figura 9.1: Ejemplos de valores de Preferencias Parciales correspondiente al cálculo de *IR* para el método "guiFebrl.MainFebrlWindow.writeStatusBar".

La Tabla 9.1 muestra ejemplos de métodos/funciones con sus respectivos valores correspondientes a *IR* y *SUD*; a través de la misma es posible analizar algunas situaciones en las que se filtran los artefactos. De los métodos/funciones exhibidos, algunos se filtraron y otros obtuvieron un valor *IR* superior al umbral, por lo que no fueron filtrados.

Signatura (reducida)	***IR***	***SUD***
classification.DecisionTree.get_classification	0 %	16 %
indexing.Indexing.get_index_stats	0 %	0 %
stringcmp.qgram	14.4 %	96 %
comparison.FieldComparator.set_weights	0 %	0 %
guiFebrl.MainFebrlWindow.writeStatusBar	0 %	100 %
dataset.DataSet.write	0 %	100 %
guiFebrl.MainFebrlWindow.runLinkCode	97.3 %	7.2 %

Tabla 9.1: Análisis de detección de utilidades para el sistema Febrl. El umbral de utilidad es de 15 %.

.

Cierto tipo de métodos, como el caso de "indexing.Indexing.get_index_stats", "comparison.FieldComparator.set_weights" y "classification.DecisionTree.get_classification" ob-

tuvieron 0 porque son métodos de acceso, incluso cuando todos tenían un valor bajo de *SUD*. Si se examinan los valores de *SUD*, estos métodos no deberían eliminarse, pero la métrica *IR* señala lo contrario. En otros casos, al igual que el método "gui-Febrl.MainFebrlWindow.writeStatusBar", el valor de *SUD* se correlaciona con *IR*, esto significa que este método se considera estructuralmente una utilidad y también obtiene el valor más bajo de *IR*; esto implica que es un método irrelevante para la lógica subyacente del sistema.

Para concluir el análisis de la etapa de filtrado, es posible observar que antes del proceso de reducción de información, el GELMF contenía 516 nodos y 1304 arcos. Luego de pasar por la fase de filtrado, el GELMF posee 132 nodos y 176 arcos. Por lo tanto, el volumen de información se redujo en proporciones sustanciales ($\sim 74\,\%$ para nodos y $\sim 86\,\%$ para arcos).

Tomando como base el GELMF resultante, que está mayormente compuesto por nodos que representan métodos/funciones relacionados con la lógica subyacente del sistema, es posible llevar a cabo el proceso de clustering con el objetivo de inferir un modelo de caso de uso del sistema. Por lo tanto, luego del proceso de reducción, todas las fases del proceso de clustering explicadas en la Subsección 7.1.2 se ejecutan automáticamente hasta la cuarta etapa (*Estrategia de Rotulado de Clusters*).

Después de que se ejecutó el proceso de clustering, el ingeniero de software debe verificar todos los nombres de los clusters y editar (o definir) un rótulo para cada cluster débilmente nombrado. En la Figura 9.2 se muestra un conjunto de clusters cuyo rótulo ya está definido y otros que no poseen uno.

Aunque una proporción de clusters no fue rotulada de manera automática, es posible etiquetar cada cluster no rotulado utilizando toda la información proporcionada por la GUI. Además, la mayoría de los clusters que se crearon a partir de los nodos manejadores obtuvieron sus etiquetas del componente interactivo de la GUI que es "manejado" por el nodo que representa el método/función manejador. Incluso cuando el cluster fue débilmente rotulado de manera automática, la información de los tooltips proporciona las mejores claves para definir el rótulo. Por ejemplo, uno de los clusters se etiquetó automáticamente como "save_button", término que fue extraído de la identificación del componente GUI en el ARGUI. Sin embargo, el texto del tooltip del botón es "Save

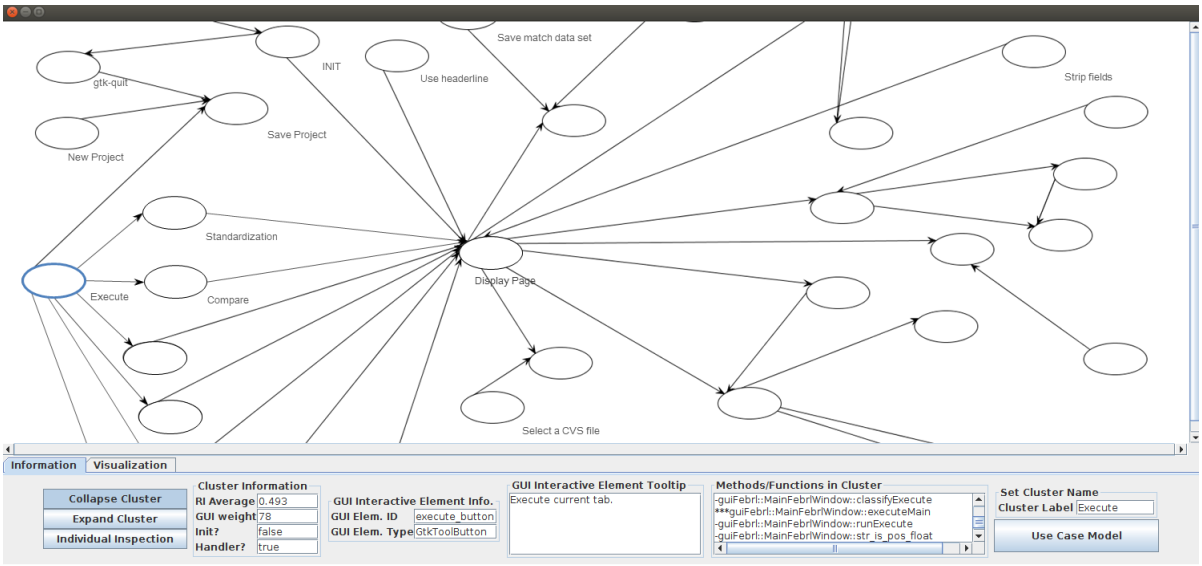

Figura 9.2: Captura de pantalla durante la fase de rotulado de clusters.

actual project in a file" (Guardar el proyecto actual en un archivo). Por lo tanto, está claro que el cluster representaba la funcionalidad "Save Project" (Guardar proyecto). Aplicando los mismos criterios, otro cluster se etiquetó como "Ejecutar", cuyo componente interactivo GUI era un botón y cuya identificación de variable en el ARGUI era "execute_button". El tooltip del botón contiene el texto "Execute current tab" (Ejecutar pestaña actual). En la mayoría de los casos, la información de los tooltips proporciona documentación esencial para determinar el nombre del cluster. Sin lugar a dudas, esta información está estrechamente relacionada con la información del Dominio del Problema, ya que está vinculada a las tareas que llevará a cabo el sistema en función de las acciones que realice el usuario a través de la interfaz gráfica.

Por otro lado, para los clusters que no se han creado en base a un nodo controlador, el criterio principal fue analizar la signatura del nodo "fuente" dentro del cluster. Por ejemplo, un cluster fue rotulado como "Compare" ya que es una dependencia de "Execute" y también estaba compuesto por un solo nodo llamado "compareExecute". Este nodo anterior hace referencia a un widget de la GUI y tiene un valor alto de *IR* e el contexto del proyecto. La estrategia genera este tipo de clusters debido a que son una dependencia exclusiva de otros clusters de manejadores y además tienen alta relevancia dentro de la lógica subyacente del sistema.

Resultados de la Comparación de Modelos

Elemento del Diagrama	#SAPSI(A)	#Gold(B)	$\|A \cup B\|$	$\|A \cap B\|$	Com(A,B)	$\%\nexists B$	$\%\nexists A$
Caso de Uso	42	37	45	35	77.8 %	6.7	15.6
$<< include >>$	49	41	50	39	78 %	4	18
Asociación	15	16	17	13	76.5 %	11.8	11.8

Tabla 9.2: Comparación entre los modelos de casos de uso ideal y obtenido con SAPSI para la herramienta Febrl.

La Tabla 9.2 muestra los resultados de la comparación del modelo obtenido para Febrl (A) y el ideal (B). En el Anexo C se pueden consultar los modelos de casos de uso `A` y `B` para Febrl. Para cada elemento del diagrama de caso de uso, la tabla muestra las siguientes variables: i) la cantidad de elementos en el modelo obtenido por SAPSI ($\#SAPSI(A)$), ii) la cantidad de elementos en el modelo ideal ($\#Ideal(B)$), iii) todos los elementos en ambos modelos ($|A \cup B|$), iv) los elementos que coinciden en ambos modelos ($|A \cap B|$), v) la proporción de coincidencias ($com(A, B)$), vi) el porcentaje de elementos en A que no fueron encontrados en B ($\%\nexists B$), y vii) el porcentaje de elementos detectados en B pero no encontrados en A ($\%\nexists A$).

Las proporciones de coincidencias son 77.8 % para casos de uso, 78 % para relaciones $<< include >>$ y 76.5 % para relaciones de asociación. Excepto por algunas diferencias, el modelo de caso de uso A es muy similar a B en cada elemento del diagrama de caso de uso de UML.

Cada caso de uso que no fue encontrado por SAPSI (A) pero apareció en el ideal (B), estaba comprendido en otro caso de uso o estaba desagregado en el modelo SAPSI. Por ejemplo, el modelo B contiene el caso de uso "Add to log" que registra todas las funciones ejecutadas en un área de texto que se visualiza en la GUI del sistema. En el modelo obtenido mediante SAPSI, todos los métodos/funciones de este caso de uso están comprendidos en el caso de uso "Display Page", que realiza las principales funciones de visualización.

Algunas diferencias en las relaciones de asociación están vinculadas con ciertos componentes dinámicos de la GUI, por lo que SAPSI detectó los casos de uso pero no detectó la asociación entre el actor y estos casos de uso. En otras situaciones, algunos casos de uso en B integran diferentes actividades que puede llevar a cabo el usuario, sin embargo en

A las mismas se encuentran desagregadas. Es decir, si bien se detectó la asociación entre el usuario y cada caso de uso desagregado, estas "nuevas asociaciones" no se encontraron en B.

Es importante resaltar el hecho de que cada caso de uso que no sea contabilizado en $A \cap B$, aumentará las proporciones $\%\sharp B$ o $\%\sharp A$, es decir, cada discordancia de casos de uso entre ambos modelos, termina incurriendo en relaciones erróneas (inexistentes) en alguno de los modelos.

A diferencia del modelo ideal para el sistema Pigeon Planner, el modelo extraído por los ingenieros de software para Febrl no contiene relaciones del tipo $<< extend >>$ o generalizaciones.

Finalmente, para tratar con la pregunta *RQ2* se aplica la medida *MoJoFM* para evaluar la efectividad del modelo de clusters obtenido de manera automática con respecto a la ideal: $MoJo(SAPSI, Ideal) = 91\,\%$. Este alto valor de similitud refuerza los resultados analizados en las comparaciones realizadas relativas a la pregunta *RQ1*.

Es importante mencionar que el caso de estudio propuesto es una aplicación ampliamente utilizada en el contexto de *data mining* (Christen, 2008). Esta herramienta presenta una GUI poco común, ya que tiene pocos botones y administra cada ejecución de la funcionalidad dependiendo de la pestaña que esté activa. Además, también genera componentes GUI de forma dinámica; por lo tanto, este tipo de componentes sólo pueden ser detectados usando TEI dinámica en tiempo de ejecución. Teniendo en cuenta que muchas características de Febrl se pueden contemplar como dificultades para el enfoque propuesto, SAPSI muestra resultados relevantes para este sistema.

La estrategia extrae un modelo de caso de uso para el sistema bajo estudio simplificando su análisis y comprensión. Es posible discernir las principales funcionalidades del sistema identificando los casos de uso más relevantes y sus dependencias.

9.2.3. Sistema Pigeon Planner

En este apartado se presentan los resultados que se obtuvieron analizar el sistema Pigeon Planner utilizando la estrategia SAPSI. En primer lugar se analizan los resultados obtenidos en la etapa de extracción, representación y filtrado de la información.

La Figura 9.3 muestra la Estructura de Agregación de SAPSI con todas las preferencias para el método "reportlib/libcairodoc.CairoDoc.paginate_document". Incluso cuando "paginate_document" obtuvo altos valores para *SUD* e *Intermediación*, el valor de IR se ve penalizado por el operador *C-+* como resultado de obtener cero para la métrica *Peso GUI*.

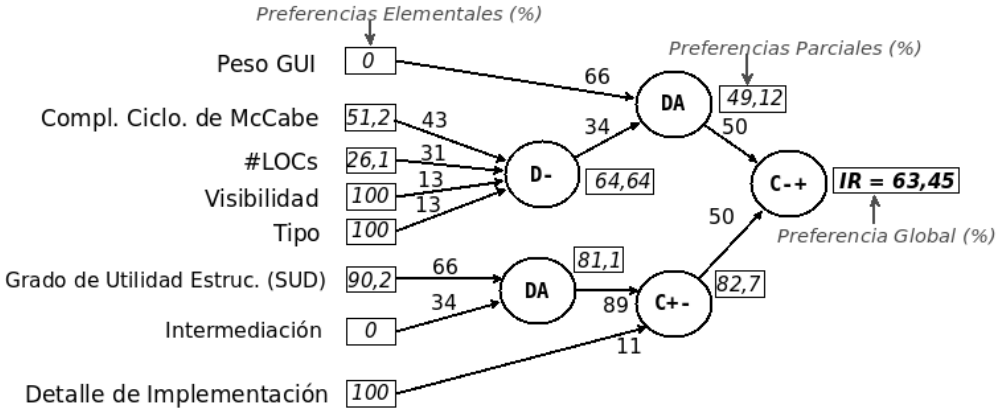

Figura 9.3: Ejemplos de valores de Preferencias Parciales correspondiente al cálculo de *IR* para el método "reportlib/libcairodoc.CairoDoc.paginate_document".

La Tabla 9.3 muestra ejemplos de métodos/funciones con sus respectivos valores correspondientes a *IR* y *SUD*; a través de la misma es posible analizar algunas situaciones en las que se filtran los artefactos. De los métodos/funciones exhibidos, algunos se filtraron y otros obtuvieron un valor *IR* superior al umbral, por lo que no fueron filtrados.

Signatura (reducida)	***IR***	***SUD***
reportlib/styles/graphicstyle.GraphicsStyle.get_line_width	0 %	19.6 %
ui/dialogs.InformationDialog.get_data	0 %	7.09 %
thumbnail.__build_path	14.7 %	19.56 %
report/pedigrees/middle.PedigreeReport.write_report	0 %	100 %
reportlib/backend/docbackend.DocBackend.close	0 %	100 %
reportlib/libcairodoc/CairoDoc.paginate_document	63.5 %	9.8 %
reports/pigeons.PigeonsReport.write_report	85.4 %	6.5 %

Tabla 9.3: Análisis de detección de utilidades para el sistema Pigeon Planner. El umbral de utilidad es de 15 %.

.

Cierto tipo de métodos, como el caso de los métodos "reportlib/styles/graphicstyle.GraphicsStyle.get_line_width" y "ui/dialogs.InformationDialog.get_data", donde ambos obtuvieron cero porque son métodos de acceso, incluso cuando tenían un valor bajo de

SUD (lo que indica que no deberían ser considerados utilidades bajo la perspectiva de dicha métrica).

En otros casos, al igual que "report/pedigrees/middle.PedigreeReport.write_report", el valor de *SUD* se correlaciona con *IR*, esto significa que el mismo es considerado un método irrelevante para la lógica subyacente del sistema.

Al finalizar la etapa de filtrado se obtienen los siguientes datos: antes del proceso de reducción de información, el GELMF contenía 1196 nodos y 2718 arcos. Luego de pasar por la fase de filtrado, el GELMF posee 374 nodos y 433 arcos. Por lo tanto, el volumen de información se redujo en buenas proporciones ($\sim 69\,\%$ para nodos y $\sim 84\,\%$ para arcos). En contraste con el caso de estudio Febrl, este sistema exhibe una GUI con más funcionalidades y esto deviene un número mayor de nodos manejadores, factor que reduce en cierta medida la cantidad de nodos filtrados.

Luego del proceso de reducción, el proceso de clustering detallado en la Subsección 7.1.2 se ejecuta de forma automática hasta la cuarta etapa donde se deben establecer los rótulos de los clusters. Durante esta etapa el ingeniero de software debe verificar todos los nombres de los clusters y editar (o definir) un rótulo para cada cluster que no posee uno o incluso redefinir aquellos que estén débilmente nombrados.

De manera similar que el sistema Febrl, en el modelo de clusters correspondiente a Pigeon Planner se encontró una proporción de clusters que no fueron rotulados de manera automática. No obstante, en la mayoría de los casos la información provista por la GUI de la herramienta posibilitó el etiquetado de cada cluster no rotulado. Por otra parte, la mayoría de los clusters que fueron creados a partir de los nodos manejadores obtuvieron sus etiquetas del componente interactivo de la GUI "manejado" por el nodo que representaba el método/función manejador. Incluso cuando el cluster no obtuvo un rótulo acorde con la funcionalidad que representa, la información de los tooltips proporcionaron frases significativas para establecer el mismo. Por ejemplo, uno de los clusters se etiquetó automáticamente como "_Restore", término que fue extraído de la identificación del rótulo del ítem del menú en el ARGUI. No obstante, el texto del tooltip del botón es "Restore a backup of your database" (Restaurar Base de dato salvada), es por esto que dicho cluster fue nombrado "Restore Database Backup". Claramente, los componentes de la GUI proporcionan información relativa al Dominio del Programa y

las distintas funcionalidades que el usuario puede llevar a cabo a través de la interacción con el sistema.

Por otro lado, para los clusters que no fueron creados en base a un nodo controlador, el criterio principal fue analizar la composición de los mismos en conjunto con las relaciones inter-clusters, es decir los clusters que dependen del mismo. En la mayoría de los casos, los rótulos de aquellos nodos que son "fuente" en cada cluster de este tipo proporcionan información para definir un rótulo acorde. En pocos casos la definición del rótulo demandó un análisis más amplio en relación al modelo de clusters, es decir, en dichos casos fue necesario inspeccionar ciertos elementos por fuera del cluster analizado. A continuación se muestran dos ejemplos de este tipo de casos:

- *Generate report:* la función fuente de este cluster era *"reportlib/basereport.report"*, la cual invocaba a un gran conjunto de métodos/funciones que componían el cluster. Si bien el nombre de la función no brindaba mucha información respecto al rol que cumplía la misma dentro del sistema, al inspeccionar el modelo era posible determinar que la funcionalidad realizaba la construcción de un reporte de manera genérica ya sea para que sea impreso o almacenado en formato PDF. Esto fue deducido a partir de que el cluster en cuestión dependía de otros dos clusters denominados "Generate PDF Interface" y "Generate Print Interface". Además este cluster era dependencia de distintas funcionalidades que estaban vinculadas con la generación de reportes y la subsiguiente impresión o generación de PDF, tales como "Preview Results", "Save Results", "Print Results", "Print Blank Pedigree", "Print Pigeons", entre otros.

- *Remove Pigeon Data:* componía distintos métodos/funciones para remover los datos de las palomas. El mismo era dependencia de funcionalidades tales como "Remove Medication", "Remove Media" y "Remove Pigeon". El cluster estaba compuesto por una función fuente con la signatura "core/pigeon.remove_pigeon", lo cual podía indicar que dicha funcionalidad permitía eliminar una paloma. No obstante, en función del análisis de las funcionalidades adyacentes, se dedujo que el la funcionalidad permitía eliminar distintos datos asociados a una paloma; es decir, no sólo implementaba la eliminación de todo el registro del ave, sino que permitía

eliminar datos específicos de forma individual.

Resultados de la Comparación de Modelos

Elemento del Diagrama	#SAPSI(A)	#Gold(B)	$\|A \cup B\|$	$\|A \cap B\|$	Com(A,B)	$\%\nexists B$	$\%\nexists A$
Caso de Uso	82	59	82	58	71 %	0	29.2
<< *include* >>	40	31	28	19	68 %	0	32.1
Asociación	65	41	65	41	63 %	0	36.5

Tabla 9.4: Comparación entre los modelos de casos de uso ideal y obtenido con SAPSI para la herramienta Pigeon Planner.

La Tabla 9.4 muestra los resultados de la comparación del modelo obtenido para Pigeon Planner (A) y el ideal (B). De forma análoga que en el caso de estudio anterior, para cada elemento del diagrama de caso de uso, la tabla muestra las siguientes variables: i) la cantidad de elementos en el modelo obtenido por SAPSI ($\#SAPSI(A)$), ii) la cantidad de elementos en el modelo ideal ($\#Ideal(B)$), iii) todos los elementos en ambos modelos ($|A \cup B|$), iv) los elementos que coinciden en ambos modelos ($|A \cap B|$), v) la proporción de coincidencias ($com(A, B)$), vi) el porcentaje de elementos en A que no fueron encontrados en B ($\%\nexists B$), y vii) el porcentaje de elementos detectados en B pero no encontrados en A ($\%\nexists A$).

Las proporciones de coincidencias fueron 71 % para casos de uso, 68 % para relaciones << *include* >> y 63 % para relaciones de asociación. En el Anexo C se pueden consultar los modelos de casos de uso `A` y `B` para Pigeon Planner. Del análisis de ambos modelos se puede inferir que el modelo de caso de uso A es muy similar a B según cada componente del diagrama de caso de uso de UML. Si bien los porcentajes de coincidencia en general resultaron más bajos en relación al caso de estudio anterior, el modelo obtenido de forma cuasi-automática es aún más preciso en relación a la evaluación de Febrl, desde el punto de vista de la apreciación del grupo de ingeniería de software. A continuación se mencionan los principales factores que afectaron los porcentajes de coincidencia:

- *Con respecto a relaciones:* a diferencia de Febrl, modelo ideal para la herramienta Pigeon Planner contiene varias relaciones del tipo << *extend* >>. Si bien no fueron consideradas a la hora de la comparación, las mismas repercuten de manera indirecta en las demás relaciones. Es decir, en la mayoría de los casos, una relación

de este tipo en el modelo ideal se mapea directamente con una << *include* >> o una asociación en el modelo de SAPSI. Por ende, este tipo de relaciones afectan los porcentajes de coincidencias en referencia a las relaciones comparadas.

- *Con respecto a casos de uso:* la discordancia entre los modelos con respecto a los casos de uso está directamente relacionada con la desagregación de distintas funcionalidades en el modelo obtenido por SAPSI. Un claro determinante de este factor son los valores encontrados en la columna que indica el porcentaje de elementos en A que no fueron encontrados en B (todos los valores son cero). En este sentido, la estrategia encontró todos los elementos en el modelo ideal, sólo que en ciertos casos los mismos han sido desagregados en más casos de uso. Este es otro factor que penaliza las coincidencias en las relaciones, ya que la desagregación de un cluster incorpora nuevas relaciones del tipo << *include* >> en el modelo obtenido por SAPSI.

Finalmente, para tratar con la pregunta *RQ2* se aplica la medida *MoJoFM* para evaluar la efectividad del modelo de clusters obtenido de manera automática con respecto a la ideal: $MoJo(SAPSI, Ideal) = 92{,}44\,\%$. Este alto valor de similitud refuerza los aspectos mencionados anteriormente respecto a las comparaciones entre modelos de casos de uso (pregunta *RQ1*).

Este caso de estudio, a diferencia de Febrl, es una aplicación más común que presenta una GUI con un funcionamiento habitual en el contexto de los sistemas de información. En este sentido, las aplicaciones con este tipo de características favorecen el proceso de análisis planteado por SAPSI. Si bien el sistema incorpora un funcionamiento sencillo, el mismo provee un gran número de funcionalidades que hacen al mismo uno de los sistemas más populares en el nicho para el cual fue desarrollado. Más allá de ciertos aspectos que penalizaron los porcentajes de coincidencias entre los modelos, es posible concluir que el modelo obtenido representa en gran medida el funcionamiento del sistema desde la perspectiva de casos de uso.

Capítulo 10

Conclusión

> El razonamiento esboza una conclusión, pero no la hace cierta hasta que la mente la descubra por el camino de la experiencia.
>
> — Roger Bacon

En este trabajo de doctorado se estudió la temática Comprensión de Programas (CP), una disciplina que asiste al ingeniero de software en las tareas de análisis y entendimiento de un sistema de software.

En la primera parte del informe se presentó dicha disciplina, describiendo sus principales propósitos e introduciendo los conceptos fundamentales asociados a la misma. Además se desarrollaron ciertas temáticas fuertemente relacionadas con el eje central de la propuesta ya que son indispensables a la hora de concebir una estrategia de comprensión efectiva. En este contexto y a través del estudio de la bibliografía referente a CP, fue posible determinar que un programador entiende un programa cuando logra vincular el Dominio del Problema con el Dominio del Programa.

Seguidamente, se presentó el estado del arte de la CP poniendo especial énfasis en el eje central del trabajo: la definición de una estrategia de CP que facilite al ingeniero de software la vinculación del Dominio del Problema con el Dominio del Programa. Por medio del análisis de la literatura relevante, fue posible destacar distintos trabajos similares al presentado desde diversas aristas. Sin embargo, también se encontraron ciertas bre-

chas en las distintas áreas que sirvieron como bases para el desarrollo de la investigación doctoral.

En el marco de la temática descripta y el estado del arte relevado, se presentó *Strategy for the Analysis of Program Static Information* (SAPSI), una estrategia de CP que asiste al ingeniero de software durante el proceso de comprensión de un sistema multiparadigma, facilitando la vinculación entre el Dominio del Problema y el Dominio del Programa. SAPSI se propuso definiendo una serie de etapas donde a medida que se va avanzando través del proceso especificado, la información es integrada progresivamente hasta llegar a una abstracción del sistema bajo estudio; en este caso en particular, se obtiene un modelo que incluye las principales funcionalidades del mismo. Es importante resaltar que este enfoque incremental de análisis e integración de la información facilita la implementación de la estrategia teniendo en cuenta los siguientes aspectos:

- los procesos relativos a la estrategia fueron agrupados en *front-end* y *back-end*, esta característica favorece la aplicación de la estrategia a distintos lenguajes de programación.

- la desagregación de las tareas en cada etapa facilita la comprensión del proceso de análisis por parte del ingeniero de software y esto simplifica la incorporación de nuevas características a la estrategia en general.

Con el objeto de mostrar las principales funcionalidades de un sistema, la estrategia permite inferir un modelo de casos de uso de UML para el mismo. En el contexto de Ingeniería de Software este tipo de modelo ha sido ampliamente utilizado con diferentes propósitos en distintos trabajos de investigación. Una de las ventajas más valiosas de emplear este tipo de modelos, es que son fáciles de entender por todos los involucrados en el desarrollo de un sistema de software y los correspondientes procesos de reingeniería.

Posteriormente se presentaron las distintas etapas de SAPSI, las cuales usan y/o implementan algunas técnicas conocidas de Ingeniería Reversa. No obstante, también se definieron nuevas técnicas o combinaciones de estas que mejoran distintos aspectos de las estrategias encontradas en el estado del arte. A continuación se expone un resumen de lo desarrollado en las etapas principales de la estrategia:

Extracción de Información: presentó la primera etapa de la estrategia, introduciendo las distintas técnicas que pueden ser utilizadas para la obtención de información en el contexto de Ingeniería Reversa. Además se expuso en detalle los procesos referentes al *front-end* de SAPSI, es decir las técnicas específicas usadas para extracción y representación de la información del sistema bajo estudio. De esta etapa se pueden derivar diversos aspectos que sirvieron como base para lo que resta de la estrategia:

- la extracción de dos representaciones con información del sistema: un grafo de llamadas específico para sistemas multiparadigmas (Grafo Estático de Llamadas a Métodos/Funciones (GELMF)) y un árbol que contiene información de la jerarquía de la GUI y sus componentes (Árbol de Representación de la GUI (ARGUI))
- la extracción de un conjunto de métricas de software orientadas a determinar la importancia de cada artefacto analizado por la estrategia.
- la combinación de la información extraída materializada en un GELMF *atribuido*, representación que servirá como base para todo el proceso de análisis llevado a cabo por SAPSI.
- el reconocimiento de otros lenguajes y librerías gráficas depende de la incorporación de los *front-ends* correspondientes.

Reducción de Información: presentó el proceso general para reducir el tamaño del GELMF atribuido, la cual es la representación de base que usa SAPSI para el análisis. Dicho proceso de reducción de información se basa en el cálculo de una métrica denominada Importancia Relativa (IR) cuyo valor representa un grado de certeza respecto de si el nodo es relevante o no en relación a la lógica subyacente del sistema. El enfoque adoptado por SAPSI es frecuentemente utilizado en el contexto de Ingeniería Reversa y requiere de la definición de un umbral que permita determinar cuando un método/función es irrelevante para el análisis. Sin embargo, no se encontraron trabajos en la literatura referente que permitan integrar distintos aspectos de un artefacto de software en particular en una única métrica de software.

SAPSI posibilita el cálculo de la métrica IR para cada método/función tomando como base distintos requerimientos que el mismo debe cumplir para ser calificado como importante. Para el cálculo hace uso de un método de evaluación denominado *Logic Scoring of Preference* (LSP).

Técnica de Clustering y Generación del Modelo de Casos de Uso: explicó la técnica de clustering utilizada por SAPSI para agrupar y abstraer los componentes del GELMF. Dicha técnica incorpora conceptos de algoritmos de clustering diseñados en el marco de CP, los cuales hacen uso de detección de ciertos patrones orientados a obtener descomposiciones del sistema que ayuden al ingeniero de software a entender el mismo. Si bien se han propuesto diversas técnicas de clustering de software con el objetivo de extraer la arquitectura del sistema bajo estudio, en esta tesis doctoral se emplearon dichas técnicas para que el modelo de clusters obtenido permita construir un modelo de casos de uso. Para esto se basó en la detección de ciertos componentes de software que están fuertemente vinculados con las funcionalidades del sistema, como lo son los componentes interactivos de la GUI y aquellos métodos/funciones que implementan el comportamiento de dicho componente. Una vez generado el modelo de clusters, SAPSI ejecuta un algoritmo de transformación para construir un modelo de caso de uso a partir del modelo de clusters obtenido previamente.

Todos los procesos descriptos en las etapas anteriores, fueron implementados en una herramienta que permite automatizar los mismos con el objeto de facilitar las tareas al ingeniero de software.

Para finalizar, se definió y se llevó a cabo una metodología que posibilita la evaluación de los resultados obtenidos con SAPSI. Para esto se inspeccionaron 2 sistemas escritos en lenguaje Python:

- *Febrl*: sistema científico que facilita distintas tareas relacionadas con *data mining* y disciplinas afines.

- *Pigeon Planner*: sistema de información para registro y asistencia durante la crianza de palomas de carrera y mensajeras,

Ambos sistemas comparten ciertas características como por ejemplo, los dos han sido desarrollados usando la librería gráfica GTK; son de tamaño mediano considerando el lenguaje de programación utilizado; ambos son de código abierto; entre otras.

De la evaluación de la estrategia se puede comprobar que en ambos casos la misma posibilitó la extracción de un modelo de casos de uso que tenía plasmadas las principales funcionalidades de cada sistema inspeccionado. En referencia a la comparación de los modelos de caso de uso extraídos en contraste con modelos ideales generados por expertos, se destaca una precisión razonable teniendo en cuenta el tipo de modelo seleccionado y las desventajas que el mismo incorpora en el contexto de Ingeniería Reversa. Por otra parte, también fue posible identificar los métodos/funciones que implementan cada caso de uso en el modelo. Teniendo en cuenta este aspecto, la precisión de los resultados obtenidos fue muy buena, indicando que la estrategia de clustering funciona de manera acertada en referencia al objetivo general planteado.

Finalmente, es posible concluir que el modelo obtenido por SAPSI facilita al ingeniero de software la interconexión entre el Dominio del Problema y el Dominio del Programa. Dicho modelo posibilita discernir las principales funcionalidades del sistema identificando los casos de uso más relevantes y sus dependencias en conjunto con aquellos artefactos del Dominio del Programa que están fuertemente relacionados con los elementos de dicho modelo.

10.1. Contribuciones

A continuación se mencionan las principales contribuciones de este trabajo de doctorado:

Una estrategia de Comprensión de Programas

Se definió una estrategia que asiste de manera efectiva con las tareas llevadas a cabo por el ingeniero de software para comprender un sistema. La misma ofrece como resultado una representación ampliamente utilizada en el contexto del desarrollo de software como lo es un modelo de caso de uso de UML. A través de esta representación el ingeniero de software logra inspeccionar las funcionalidades que provee el sistema pudiendo navegar

los artefactos de software que sirven para implementar las mismas. De esta forma, se provee un modelo del sistema que brinda información referente a las funcionalidades que este provee y los artefactos de software usados para implementar las mismas.

Extracción de Información y Construcción de Representaciones

Se construyeron dos representaciones específicas para SAPSI (GELMF y ARGUI) a partir de la información extraída del sistema bajo estudio . La estructura principal, el Grafo Estático de Llamadas a Métodos/Funciones (GELMF), es utilizada de manera frecuente en el contexto de Ingeniería Reversa, sin embargo la estructura que posee información de la GUI (ARGUI) no es usada frecuentemente en este contexto y en mucho menos en combinación con la anterior. Por otra parte, también se extrajo/computó un conjunto de métricas de software que caracterizan los artefactos analizados (métodos/funciones) desde el punto de vista de la importancia relativa que cada uno de estos tiene en referencia a todo el sistema. La integración de ambas representaciones y en conjunto con las métricas brindan tres aspectos importantes de los dominios de interés para la estrategia: i) el GELMF contiene las relaciones entre cierto tipo de artefactos esenciales del código fuente (métodos/funciones), ii) el ARGUI está fuertemente vinculado con algunos componentes del Dominio del Problema, iii) las métricas proporcionan características que permiten valorar los elementos considerados en el GELMF desde distintos puntos de vista. Finalmente, toda la información es integrada en el GELMF, construyendo un grafo atribuido con toda la información requerida para llevar a cabo el proceso de CP.

Definición de la Métrica *Importancia Relativa*

Tomando como base las representaciones mencionadas en el apartado precedente, se definió una métrica denominada Importancia Relativa (IR), la cual permite estimar la relevancia de un método/función respecto de la lógica subyacente del sistema. Para el cálculo de la misma es necesario tener en cuenta ciertos aspectos inherentes a cada artefacto de software "medido", como por ejemplo las llamadas entre métodos/funciones o la relación de cada artefacto con la GUI del sistema. Los valores asociados a la medición de todos estos aspectos se deben combinar para obtener un valor que sirva como indi-

cador de relevancia del artefacto de software. Las características antes mencionadas se ajustan a la definición de métodos multicriterio. Es por este motivo que para el cálculo de esta métrica en particular se utiliza el método LSP. Dicho método permite agregar los valores de distintas métricas mediante el uso de operadores lógicos. Esto posibilita la obtención de un único valor que se verá afectado por la medición de distintos aspectos que caracterizan a cada artefacto analizado. La métrica IR se usa para filtrar información irrelevante y para asistir al proceso de clustering.

Especificación de los Elementos que Requiere el Método LSP

Para obtener la Preferencia Global (IR), LSP requiere de la definición de ciertas estructuras, específicamente Árbol de Criterios, Estructura de Agregación y Criterios Elementales. Si bien las estructuras de base pueden variar con el tiempo, se han desarrollado todos los componentes que requiere LSP para evaluación de un elemento en particular. En este caso, se diseñaron para evaluar la importancia relativa de un artefacto de software, específicamente un método/función, dentro del contexto de la lógica subyacente de un sistema. Cabe destacar que la flexibilidad de LSP y el proceso de filtrado hacen evidente la posibilidad de calibrar todo el modelo de acuerdo a nuevos requerimientos ya sea porque fueron detectados durante el proceso de Ingeniería Reversa o porque fueron impuestos por el mismo. Claramente, esto se puede hacer adaptando los parámetros LSP de acuerdo a los cambios requeridos. La flexibilidad del método permite que este procedimiento de calibración sea conducido principalmente por la experiencia del ingeniero de software.

Definición de un Método de Filtrado Basado en la Métrica Importancia Relativa

Para reducir el tamaño de la estructura de base que usa la estrategia para llevar a cabo el análisis (GELMF atribuido) se propuso un algoritmo de filtrado de información que toma como base la métrica antes descripta. El algoritmo considera como principal criterio la métrica IR y un umbral que determina cuando un artefacto en particular debería ser considerado de poca importancia para la lógica subyacente del sistema.

Diseño de una técnica de Clustering Orientada a la Comprensión de Programas

Para generar la abstracción del sistema que facilite la vinculación entre los Dominio del Problema y del Programa, la estrategia genera como resultado un modelo de casos de uso de UML. Para esto se definió una técnica de clustering de software orientada a la Comprensión de Programas que facilita la obtención del tipo de modelo referenciado anteriormente. El algoritmo toma como principal criterio de agrupamiento la siguiente información: i) los componentes de la GUI del sistema, ii) las conexiones de estos componentes con el código fuente y iii) la métrica IR previamente mencionada. Dentro de estos tres elementos, los primeros están estrechamente vinculado con el Dominio del Problema de la aplicación, el segundo sirve como medio para construir la vinculación entre ambos Dominios y el tercero posibilita cuantificar la importancia de cada artefacto de software para con el sistema.

Implementación de SAPSI en una herramienta

Con el objetivo de materializar todas las técnicas y conceptos integrados en SAPSI, se desarrolló *Dupin*, una herramienta de Ingeniería Reversa que asiste al ingeniero de software en la vinculación de los Dominios. Uno de los aspectos más relevantes de *Dupin* es que la arquitectura presenta una distribución en capas, donde se puede distinguir, en la desagregación de más alto nivel, al *front-end* y al *back-end* como las dos principales. La primera agrupa aquellos módulos que extraen información del sistema bajo estudio y construyen la representaciones de base que serán usadas durante todo el proceso de análisis. Esta capa está fuertemente relacionada con las características relativas a la plataforma y las tecnologías usadas por el sistema, es decir, los aspectos cercanos a las construcciones sintácticas del lenguaje de programación, las librerías gráficas empleadas, otras características del lenguaje, entre otros. La capa del back-end es independiente del lenguaje utilizado e implementa la parte esencial de la estrategia. Este tipo de arquitecturas de herramientas de CP permiten aplicar la estrategia a distintos lenguajes de programación y de esta forma abarcar una gama más amplia de sistemas. De esta manera, el reconocimiento de otros lenguajes de programación y librerías gráficas se reduce

al desarrollo de los front-ends para los mismos.

Obtención de un Modelo que facilite la Vinculación de Dominios

Se obtuvo un modelo que permitió en cierta manera vincular determinados componentes del Dominio del Problema con artefactos de software del Dominio del Programa. Para esto se eligió el modelo de caso de uso de UML, el cual es ampliamente utilizado en distintas instancias del proceso de desarrollo de software.

Diseño de una Metodología para la Evaluación de los Resultados Obtenidos

Debido a que en la literatura disponible no se encontraron técnicas efectivas para comparar dos modelos de casos de uso en el contexto de Ingeniería Reversa, se definió una metodología que permite llevar a cabo esta comparación y de esta forma poder evaluar los resultados de SAPSI. En este contexto, la metodología se puede resumir mediante los siguientes pasos:

- Se reúne un grupo de ingenieros de software para llevar a cabo las tareas de Ingeniería Reversa. Se los separa en grupos pequeños para que trabajen de manera independiente y así se generen distintos modelos que posteriormente converjan en un único modelo ideal.

- Se les solicitó que generen un modelo de casos de uso del sistema analizado haciendo uso de la documentación que el mismo presente y la técnica de *análisis comportamental*.

- Luego se proporciona una herramienta de captura de trazas que posibilitó al grupo de ingenieros asignar los métodos/funciones del código fuente a cada caso de uso extraído.

- Se realizaron distintas sesiones de trabajo entre todos los grupos para consensuar un único modelo ideal correspondiente al sistema bajo estudio.

- Finalmente se comparó el modelo de casos de uso ideal con el obtenido por SAPSI teniendo en cuenta cada elemento que presenta el diagrama de casos de uso, es decir

se analizaron las coincidencias respecto de casos de uso, relaciones << *include* >> y relaciones de asociación. Esta tarea la desarrolló con la supervisión del grupo de ingenieros de software ya que las comparaciones requieren de un análisis pormenorizado de ambos modelos.

- Para la comparación de los modelos de cluster, se utilizó una métrica ampliamente conocida en la disciplina denominada *MoJo*.

10.2. Discusión y Conclusión

En esta tesis doctoral se propuso interconectar el Dominio del Problema con el Dominio del Programa en sistemas desarrollados en lenguajes multiparadigma. Para alcanzar este objetivo se presentó una estrategia de CP que mediante el uso de técnicas de Extracción y Representación de Información, Reducción de Información, Clustering, Evaluación Multicriterios, entre otras, permite generar un modelo de casos de uso del sistema bajo estudio. Este modelo tiene la particularidad de destacar las principales funcionalidades que el usuario puede realizar mediante la interacción con el sistema. Es por este motivo que el mismo está estrechamente relacionado con el Dominio del Problema, ya que refleja las funcionalidades que el sistema implementa, las acciones que el usuario puede realizar mediante el uso del sistema y cómo todas las funcionalidades (tanto aquellas que se ejecutan por orden del usuario como las que dependen indirectamente de las acciones del usuario) se relacionan entre sí. Además, la estrategia presentada posibilita la identificación de los componentes del código fuente que implementan cada caso de uso del modelo. De esta manera, el ingeniero de software no sólo cuenta con un modelo que brinda una vista interesante del sistema la cual permitirá entender distintos aspectos del funcionamiento del mismo, sino que, además permite consultar aquellos componentes de software que sirven para implementar cada uno de los artefactos que componen dicha vista.

Para el contexto de CP, el modelo de casos de uso posee ciertas ventajas que fueron mencionadas en párrafos anteriores, no obstante, las mismas ventajas se presentan como dificultades a la hora de intentar extraer este tipo de modelos. El motivo principal de este factor es que el modelo en cuestión es generado en las primeras etapas de diseño y

sirve como medio de comunicación entre los involucrados en el desarrollo. Además, del análisis de los elementos en el diagrama de casos de uso, se puede determinar que los mismos no están directamente vinculados con artefactos de código fuente, en contraste con otros modelos, como por ejemplo los diagramas de clases, diagramas de secuencia, diagramas de objetos, etc. Es por este motivo que esta clase de modelos a construir siempre representarán desafíos importantes a la hora de extraerlos de manera *cuasi-automática*. Como ejemplos de otros modelos similares es posible identificar al modelo de *Business Process Modeling and Notation* (BPMN), las ontologías o mapas de casos de uso.

Como se planteó en el objetivo principal de esta tesis, la estrategia de CP desarrollada a lo largo del trabajo y todos los conceptos usados en su elaboración muestran un método para relacionar los Dominios del Problema y del Programa en sistemas desarrollados usando lenguajes con soporte multiparadigma. Para sustentar esta afirmación, se realizaron las siguientes tareas:

- Se implementó la estrategia SAPSI en *Dupin* (Capítulo 8), una herramienta de CP que permite extraer de forma cuasi-automática un modelo de casos de uso del sistema. Dicha herramienta fue desarrollada específicamente para verificar la estrategia propuesta.

- Se utilizó *Dupin* para inspeccionar dos sistemas Python denominados *Febrl* y *Pigeon Planner* (ver Capítulo 9). Con estos experimentos se concluyó que, a pesar de los distintos niveles de complejidad presentados por las aplicaciones analizadas, es posible relacionar los Dominios del Problema y Programa mediante la inspección del modelo obtenido, ya que el mismo posibilita el análisis de las funcionalidades provista por el sistema en función del nivel de abstracción más conveniente.

De la evaluación de la estrategia se puede comprobar que el modelo obtenido por SAPSI se asemeja en gran proporción al generado por el grupo de ingenieros de software. Esto indica que SAPSI posee un grado de efectividad conveniente a la hora de proveer un modelo de casos de uso para el sistema bajo análisis. Además también muestra que es efectiva para relacionar los métodos/funciones más importantes con los casos de uso extraídos. Los resultados indican que la relación entre los Dominios del Proble-

ma y Programa ayuda a identificar rápidamente las componentes del programa usadas para implementar cada funcionalidad particular. Por esta razón, se puede deducir que una herramienta que integre este tipo de estrategias de CP reduce los costos y tiempo implicados en el mantenimiento y evolución de software; ya que el ingeniero de software se concentra solamente en el análisis de las funciones relacionadas con el componente del sistema que se desea modificar.

Teniendo en cuenta los factores mencionados en párrafos previos, la estrategia de comprensión propuesta en esta tesis ofrece distintas contribuciones y obtiene resultados relevantes en el contexto de una disciplina que se encuentra cada vez más en auge debido a las demandas del entorno. Si bien existen diversas propuestas con características similares, incluso aun más robustas y eficaces que la presentada en esta tesis, también se puede destacar cierto desinterés por parte de la comunidad científica para con ciertas temáticas relevantes que engloba la disciplina. Un claro ejemplo de este factor es la falta de propuestas de clustering orientadas a la CP, es decir, existen numerosas propuestas de clustering de software, pero pocas que impulsen la interconexión entre Dominios.

10.3. Trabajos Futuros

SAPSI tiene el potencial de ser ampliada para desarrollar una estrategia más sólida, potente y práctica para comprender los sistemas de software mediante la interconexión de dominios. Esto requiere guiar la investigación actual hacia las siguientes líneas:

Permitir el análisis de información dinámica: La estrategia utiliza Técnicas de Extracción de Información (TEI) estática para obtener información del sistema y construir las representaciones de base, específicamente, el ARGUI y el GELMF. Aunque la mayoría de las dependencias se pueden detectar utilizando estas técnicas, se sabe comúnmente que las dependencias dinámicas podrían desempeñar un papel importante en determinados tipos de aplicaciones. Además, ciertos sistemas generan componentes de la GUI en tiempo de ejecución y no se pueden detectar con información estática. Por lo tanto, una de las extensiones que mejorarían la precisión de la estrategia es la incorporación de TEI dinámica. Como ejemplo de

este tipo de extensiones se puede considerar la extracción de información del comportamiento del sistema (por ejemplo, mediante el uso de trazas de ejecución) para validar y complementar la información extraída empleando TEI estática. Este tipo de información, además de mejorar la precisión de la información plasmada en las representaciones (ARGUI y GELMF), también puede resultar sustancial en un enfoque relacionado con el uso de modelos de casos de uso. El motivo principal es que los mismos están fuertemente relacionados con los escenarios que se establecen para obtener las trazas de ejecución.

Detectar todos los componentes del modelo de casos de uso: El modelo de casos de uso generado por SAPSI está limitado a ciertos componentes del diagrama, específicamente a casos de uso, dependencias del tipo << *include* >>, relaciones de asociación y un actor. Si bien, por lo general, los modelos están compuestos mayoritariamente por este tipo de elementos, el hecho de no detectar todos los componentes es un limitante a la propuesta. Hasta el momento, la estrategia no detecta los distintos actores que puede tener un sistema (sólo identifica el usuario principal como único actor), ni relaciones del tipo << *extend* >> ni tampoco generalizaciones. Se propone extender la estrategia, específicamente la etapa de generación de modelo de casos de uso, mediante una técnica de análisis de relaciones para poder identificar relaciones << *extend* >> y generalizaciones. Además se prevé agregar heurísticas de detección de patrones, para identificar los distintos tipos de actores que puede tener el sistema.

Definir perfiles de sistemas: como se puede deducir, el sistema requiere de ciertos umbrales y valores relacionados con LSP que varían de acuerdo a las características del sistema bajo estudio, por ejemplo, el tamaño del mismo, si es móvil o basado en la web, si usa servicios web, entre otros. Es por este motivo que sería importante definir perfiles que caracterizan los diferentes tipos de sistemas y también proporcionar la configuración y los umbrales que deberían usarse con cada tipo de sistema.

Extraer otros modelos: el tipo de información que SAPSI extrae del sistema es muy importante y puede ser utilizada para generar otro tipo de modelos. En este sentido

se prevé agregar nuevas "sub-estrategias" a SAPSI para que puede generar otro tipos de vistas, modelos, representaciones que permitan fortalecer la interconexión obtenida hasta el momento. Si bien el modelo de casos de uso brinda una vista fundamental para comprender el sistema, la obtención de otros tipos de modelos proporcionará al ingeniero de software distintas perspectivas para fortalecer la estrategia. Por ejemplo, modelos de clases, diagrama BPMN, ontologías, modelo de secuencia, modelo de actividades, entre otros.

Definir mecanismos de abstracción para los modelos obtenidos: Se pretende brindar otros mecanismos extra de abstracción de los modelos obtenidos con el objetivo de facilitar la navegación y visualización por parte ingeniero de software. Esto es relevante ya que para aquellos sistemas de importante envergadura, los modelos obtenidos pueden llegar a ser voluminosos. Es por esto que sería útil incorporar mecanismos que abstraigan dichos modelos de manera tal de facilitar la inspección. Un claro ejemplo de esto sería utilizar la jerarquía del ARGUI para agrupar aquellos casos de uso originados a partir de métodos/funciones manejadores. Por ejemplo, si tres botones ("alta", "baja" y "modificación") se encuentran en una pestaña rotulada ("ABM Usuario"), sería posible agregar las funcionalidades individuales en una más abstracta usando como rótulo el título de la pestaña.

Integrar un navegador de modelos a *Dupin*: la herramienta visualiza el GELMF atribuido y permite al usuario navegar el mismo en busca de la información. Sin embargo, el modelo de casos de uso generado se exporta en un archivo que debe ser visualizado con una herramienta externa. Debido a limitaciones del formato del archivos, el modelo generado no contiene toda la información relevante. Es por este motivo que se pretende incorporar un navegador de modelos a *Dupin* para que la herramienta quede *autocontenida* y además se pueda navegar el modelo resultante con toda la información integrada, de la misma manera que es posible navegar el GELMF atribuido.

Anexos

Anexo A

Ejemplo de Análisis con SAPSI: Libreta de Contactos

Dar ejemplo no es la principal manera de influir sobre los demás; es la única.

—Albert Einstein

En este anexo se presenta un ejemplo completo mostrando la información manipulada a través de las distintas etapas de la estrategia. Para esto se utiliza SAPSI con el objeto de analizar una "aplicación de juguete" que implementa una libreta de contactos básica en el lenguaje Python. Dicha aplicación permite almacenar una lista de contactos donde por cada contacto se almacena Nombre, Dirección y Teléfono. La información almacenada se guarda en un archivo de texto el cual funciona como una "base de datos". La aplicación permite agregar, buscar, borrar y navegar a través de un listado de contactos.

En las siguientes secciones se exhibe el proceso completo que lleva a cabo SAPSI para analizar un sistema en particular:

- En primer lugar se inspeccionan los elementos que componen la información que la estrategia toma como base para realizar el análisis, en conjunto con la construcción de las representaciones de base y las métricas que va a requerir la capa del back-end.
- En segundo lugar se analizan los aspectos relacionados con el filtrado de informa-

ción, específicamente la información de la métricas que permiten calcular la IR y las información de las distintas estructuras de LSP que facilitan este cómputo.

- Por último se muestra el modelo de cluster obtenido en conjunto con el modelo de casos de uso UML de la aplicación analizada.

A.1. Extracción de Información

Como se mostró en el Capítulo 5, en esta etapa se analizan los artefactos del código fuente a partir de los cuales se extrae la información que se toma como punto de partida para el análisis y además se muestran las representaciones construidas que serán analizadas durante todo el proceso planteado por la estrategia.

A.1.1. Recursos fuentes (archivos)

En este apartado se analizan los recursos a partir de los cuales la estrategia SAPSI recolecta información para construir las representaciones de base que son utilizadas durante todo el proceso de análisis. El front-end específico desarrollado para las pruebas permite analizar aplicaciones escritas en lenguaje Python y la librería gráfica GTK. Para una mejor visualización y análisis de los recursos, lo mismos se presentan en el Anexo D.

A.1.2. Grafo Estático de Llamadas a Métodos/Funciones (GELMF)

En la Figura A.1 se exhibe el grafo obtenido mediante las técnicas descriptas en el Capítulo 5 para la aplicación analizada en este anexo. Los identificadores de los métodos/funciones se encuentran separados por la cadena "::" para poder identificar de mejor forma los componentes de la signatura de cada artefacto, específicamente archivo, clase (en caso de corresponder) y nombre de método/función).

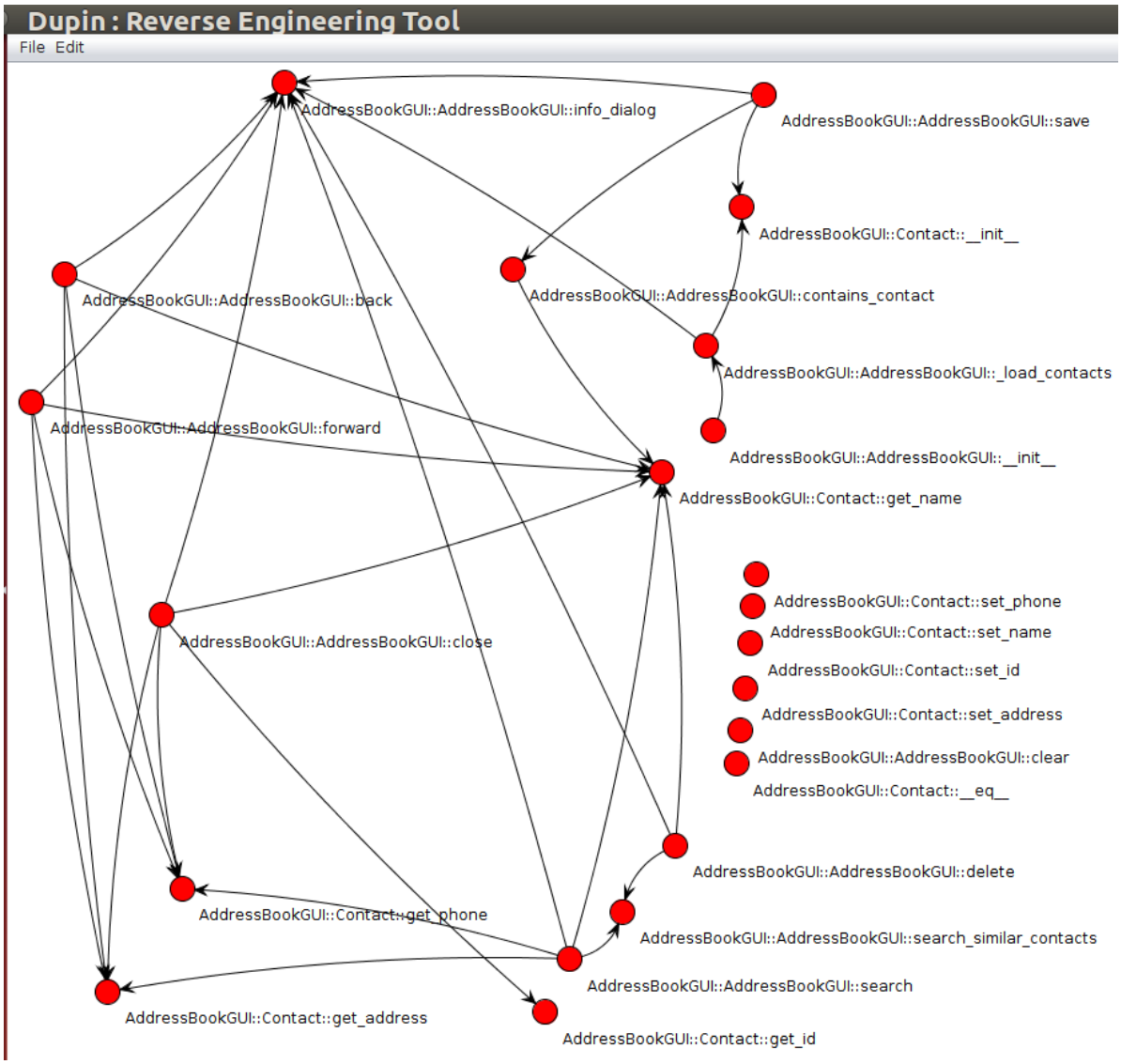

Figura A.1: Visualización del GELMF correspondiente a la aplicación libreta de contactos construido con la herramienta *Dupin*

A.1.3. Árbol de Representación de la GUI (ARGUI)

La interfaz gráfica de la aplicación analizada se puede visualizar en la Figura A.2. La misma es generada por el archivo glade (XML) exhibido anteriormente. En la misma figura también se visualiza el ARGUI (resumido) generado a partir de la estructura de la GUI de la aplicación. Esta estructura representa de manera simplificada la interfaz gráfica de la aplicación bajo análisis.

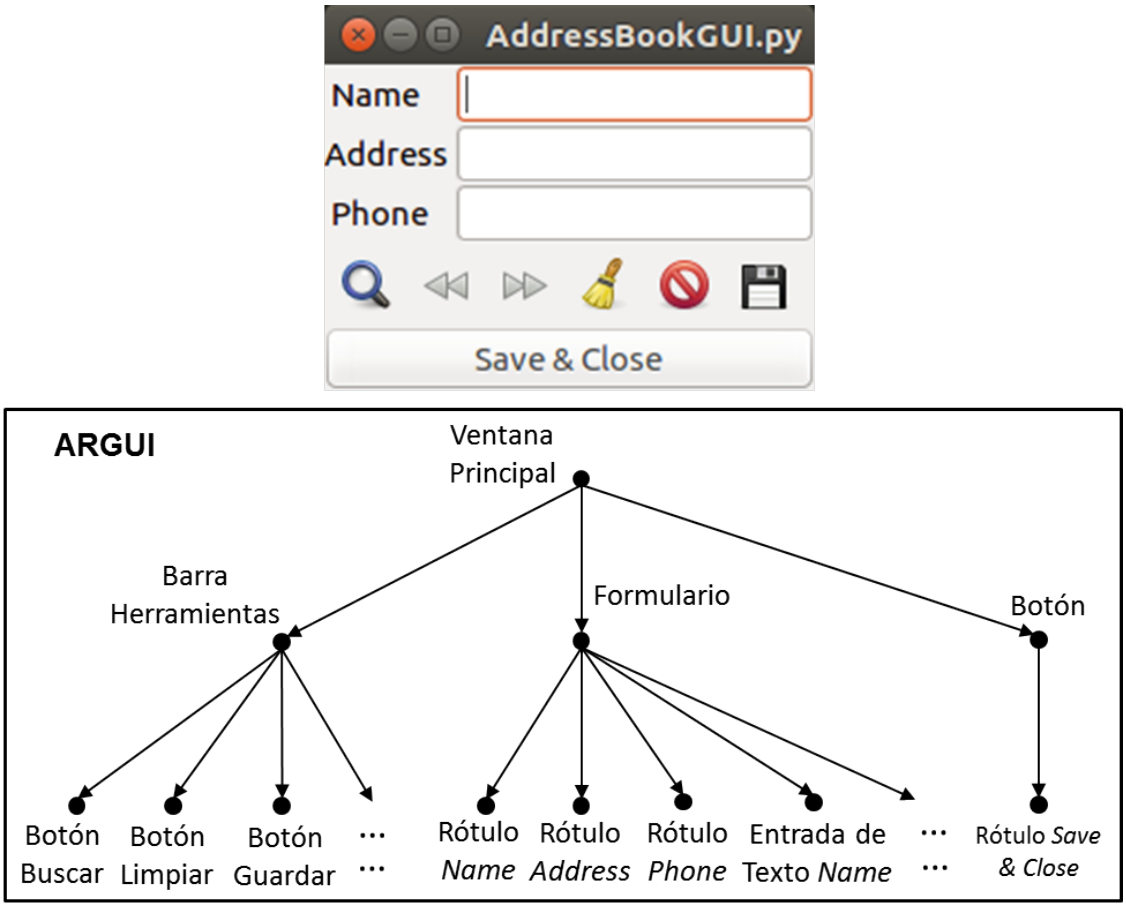

Figura A.2: GUI y visualización del ARGUI (resumido) generado a partir de un archivo glade con la estructura de la GUI de la aplicación

A.1.4. Representación de las Vinculaciones entre estructuras

En la Figura A.3 se representa como se vinculan ambas estructuras indicando el tipo de relación. Como se puede observar, se han resaltado las vinculaciones entre ciertos widgets de la GUI y dos métodos del GELMF. En color naranja se representan los vínculos del tipo "manejador", es decir qué componente de la GUI es manejado por el nodo en el GELMF; mientras que en color celeste se representan los del tipo "uso", es decir, qué widgets son usados, ya sea para lectura o edición, por cada método/función. Teniendo en cuenta lo antes descripto, se puede afirmar que el método `save` perteneciente a la clase `AddressBookGUI` definida en el archivo `AddressBookGUI.py`, implementa el comportamiento del botón con el icono que denota `guardar`. Además, este método hace uso de los siguientes widgets: las entradas de texto para los campos `name`, `phone` y

`address`; en conjunto con los botones que representan `adelantar` y `retroceder` dentro del conjunto de contactos buscados. El mismo análisis se puede realizar para el método `back` perteneciente a la misma clase.

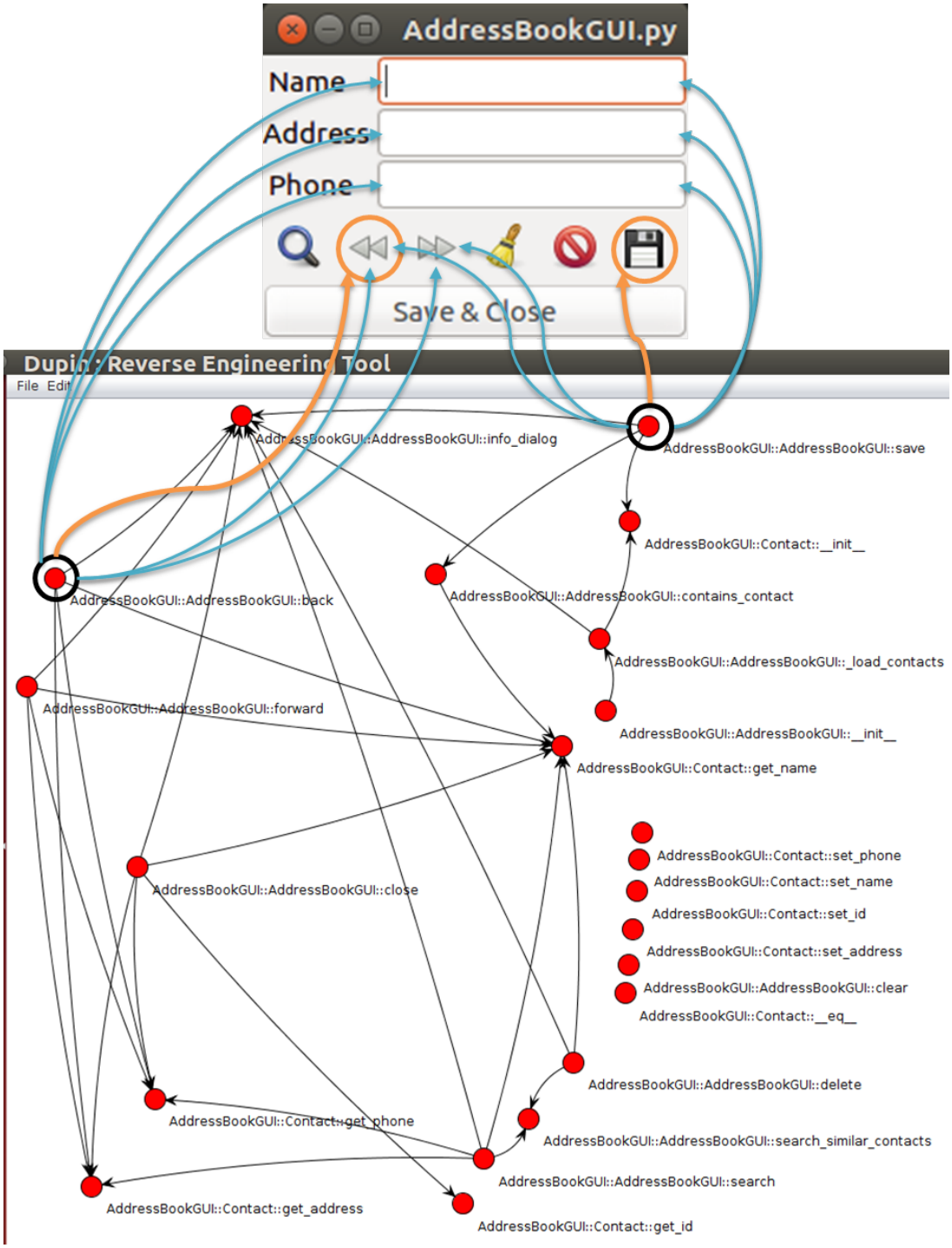

Figura A.3: Representación de la vinculación entre elementos del GELMF y la GUI de la aplicación

Estas vinculaciones son identificadas con ayuda del analizador sintáctico teniendo en cuenta ciertas sentencias de acuerdo al tipo de vinculación:

Manejadores de widget: se analizan las sentencias que conectan las señales del widget con algún método/función de la GUI. En el caso del ejemplo planteado en este apartado, se puede observar la sentencia `connect_signals(dic)`[1] que permite vincular los widgets interactivos con sus respectivos manejadores. Si bien en este caso se usa un diccionario para conectar a todos los manejadores con los componentes, este proceso se suele llevar a cabo de manera individual, conectando cada componente con una señal. También existe otro mecanismo de vinculación donde en vez de diccionarios se utiliza la propia clase como manejador, en este caso el nombre de cada método de la clase que coincida con la señal de cada widget será quien maneje el comportamiento del mismo. De esta manera se evita el uso de diccionarios.

Lectura/Edición widget: se detecta cualquier tipo de sentencia que "invoque" un widget específico de la GUI. Para el caso del ejemplo expuesto, dichas sentencias son `get_object`; por ejemplo, en el método `save` se puede identificar la sentencia `get_object("forward_button")` [2]. La misma permite acceder al botón de la GUI que sirve para retroceder en la navegación de los contactos.

A.1.5. Extracción de Métricas

Como se explicó en el Capítulo 5, para la extracción de las distintas métricas se utiliza las representaciones extraídas en pasos previos en conjunto con algunos scripts que facilitan la obtención de cierta información especifica. En la Tabla A.1 se detalla el listado de los métodos/funciones que componen el GELMF y los valores obtenidos para las métricas consideradas por la estrategia:

[1] `self.builder.connect_signals(dic)`
[2] `forward_button=self.builder.get_object("forward_button")`

Nombre método/función	Peso GUI	CC[a]	#LOCS[b]	Vis.[c]	Tipo	SUD[d]	BC[e]	DI[f]
AddressBookGUI.AddressBookGUI.info_dialog	1	2	7	si	si	1	0	no
AddressBookGUI.AddressBookGUI.back	11	2	18	si	si	0	0	no
AddressBookGUI.AddressBookGUI.close	0	3	14	si	si	0	0	no
AddressBookGUI.AddressBookGUI.__init__	3	1	24	si	si	0	0	no
AddressBookGUI.AddressBookGUI.forward	11	2	18	si	si	0	0	no
AddressBookGUI.AddressBookGUI.save	11	3	20	si	si	0	0	no
AddressBookGUI.AddressBookGUI.contains_contact	0	3	7	si	si	0.17	0.67	no
AddressBookGUI.AddressBookGUI._load_contacts	0	4	19	no	si	0.14	1	no
AddressBookGUI.AddressBookGUI.search	11	3	24	si	si	0	0	no
AddressBookGUI.AddressBookGUI.clear	11	1	13	si	si	1	0	no
AddressBookGUI.AddressBookGUI.search_similar_contacts	0	3	7	si	si	0.45	0	no
AddressBookGUI.AddressBookGUI.delete	3	5	15	si	si	0	0	no
AddressBookGUI.Contact.get_phone	0	1	2	si	si	0.90	0	si
AddressBookGUI.Contact.get_name	0	1	2	si	si	1	0	si
AddressBookGUI.Contact.__init__	0	1	5	si	si	0.45	0	si
AddressBookGUI.Contact.set_address	0	1	2	si	si	1	0	si
AddressBookGUI.Contact.set_name	0	1	2	si	si	1	0	si
AddressBookGUI.Contact.get_id	0	1	2	si	si	0.22	0	si
AddressBookGUI.Contact.__eq__	0	3	3	si	si	1	0	si
AddressBookGUI.Contact.set_id	0	1	2	si	si	1	0	si
AddressBookGUI.Contact.get_address	0	1	2	si	si	0.9	0	si
AddressBookGUI.Contact.set_phone	0	1	2	si	si	1	0	si

Tabla A.1: Extracción de métricas para el ejemplo libreta de contactos

[a] CC= Complejidad Ciclomática [b] #LOCS= Número de Líneas de Código [c] Vis.= Visibilidad
[d] SUD= Structural Utility Degree [e] DI= Detalle de Implementación [f] BC= Betweenness Centrality

A.2. Reducción de la Información

En esta sección se muestra el resultado de llevar a cabo el proceso de filtrado definido por SAPSI aplicado a la aplicación analizada en este anexo.

A.2.1. Obtención de Preferencias Elementales

La Tabla A.2 presenta las Preferencias Elementales correspondiente a las métricas consideradas por SAPSI para cada método/función extraído del ejemplo. Como se explicó en el Capítulo 6, dichas preferencias han sido obtenidas al aplicar las Funciones de Criterio Elemental especificadas en la Sección 6.3 a las Variables de Preferencia de cada criterio (ver Tabla A.1).

Es importante recordar que para las Funciones de Criterio Elemental $g_{p_{gui}}$, g_{CC} y $g_{\#LOCs}$ se deben especificar los valores x_{min} y x_{max} (ver Sección 6.3), a continuación se especifican los valores extremos que se usaron para evaluar el ejemplo analizado:

$g_{p_{gui}}$: se asigna $x_{max} = 5$ ya que es un valor que representa cuando el método/función ha interactuado con un conjunto de widgets con un peso que debe ser tenido en cuenta. El valor de $x_{min} = 0$ ya que es el menor valor que puede tomar un nodo y representa la situación en que el mismo no interactúa con ningún widget de la GUI.

- x_{min}: 0
- x_{max}: 5

g_{CC} El algoritmo toma como x_{max} el promedio de los valores de CC de todos los nodos, considerando aquellos que cumplen $CC > 1$. Como se puede observar, el sistema no presenta una complejidad elevada y por ende la complejidad promedio es un valor bajo. Por definición de la métrica, el menor valor de complejidad que puede obtener un método/función es 1.

- x_{min}: 1
- x_{max}: 3

$g_{\#LOCs}$ la pauta para seleccionar los valores extremos para este Criterio Elemental es el mismo que para CC. El algoritmo toma como x_{max} el promedio de los valores de $\#LOCs$ de todos los nodos, considerando aquellos que cumplen $\#LOCs > 2$ ya que cualquier método o función en Python posee al menos 2 líneas de código[3].

- x_{min}: 2
- x_{max}: 13.86

A.2.2. Obtención de Preferencias Globales para los Nodos del GELMF

En este apartado se muestra el resultado del cálculo de la métrica IR para los nodos del GELMF del ejemplo analizado. En la Tabla A.3 se muestran las Preferencias Parciales obtenidas por cada operador y la Preferencia Global que representa la métrica calculada. Las columnas de la tabla se muestran en orden de resolución de los operadores CDG de acuerdo a la Estructura de Agregación que utiliza la estrategia. Con propósitos de facilitar el análisis de los valores que presenta la tabla, en la Figura A.4 se visualiza la Estructura de Agregación definida para SAPSI.

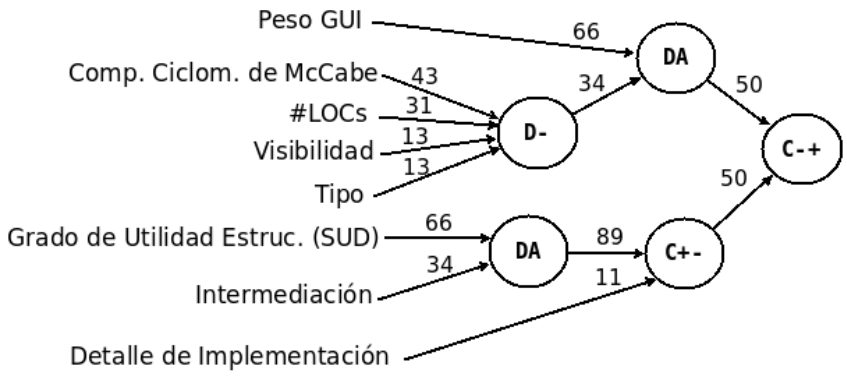

Figura A.4: Estructura de Agregación para computar IR

Como se puede deducir a partir del análisis de la tabla con las Preferencias Parciales y Globales, una gran cantidad de métodos[4] obtienen cero como Preferencia Global (valor que representa la IR). Varios métodos de la clase `Contacto` son clasificados como

[3]Esto se debe a que usa la indentación como su modo de separación de bloques de sentencias.
[4]En este caso en particular, la aplicación no posee funciones

Nombre método/función	Peso GUI	CC	*#LOCS*	Vis.	Tipo	SUD	BC	DI
AddressBookGUI.AddressBookGUI.info_dialog	0.2	0.67	0.5	1	1	0	0	1
AddressBookGUI.AddressBookGUI.back	1	0.67	1	1	1	1	0	1
AddressBookGUI.AddressBookGUI.close	0	1	1	1	1	1	0	1
AddressBookGUI.AddressBookGUI.__init__	0.6	0	1	1	1	0	0	1
AddressBookGUI.AddressBookGUI.forward	1	0.67	1	1	1	1	0	1
AddressBookGUI.AddressBookGUI.save	1	1	1	1	1	1	0	1
AddressBookGUI.AddressBookGUI.contains_contact	0	1	0.5	1	1	0.83	0.67	1
AddressBookGUI.AddressBookGUI._load_contacts	0	1	1	0	1	0.85	1	1
AddressBookGUI.AddressBookGUI.search	1	1	1	1	1	1	0	1
AddressBookGUI.AddressBookGUI.clear	1	0	0.94	1	1	0	0	1
AddressBookGUI.AddressBookGUI.search_similar_contacts	0	1	0.5	1	1	0.55	0	1
AddressBookGUI.AddressBookGUI.delete	0.6	1	1	1	1	1	0	1
AddressBookGUI.Contact.get_phone	0	0	0	1	1	0.10	0	0
AddressBookGUI.Contact.get_name	0	0	0	1	1	0	0	0
AddressBookGUI.Contact.__init__	0	0	0.36	1	1	0.55	0	0
AddressBookGUI.Contact.set_address	0	0	0	1	1	0	0	0
AddressBookGUI.Contact.set_name	0	0	0	1	1	0	0	0
AddressBookGUI.Contact.get_id	0	0	0	1	1	0.77	0	0
AddressBookGUI.Contact.__eq__	0	1	0.22	1	1	0	0	0
AddressBookGUI.Contact.set_id	0	0	0	1	1	0	0	0
AddressBookGUI.Contact.get_address	0	0	0	1	1	0.1	0	0
AddressBookGUI.Contact.set_phone	0	0	0	1	1	0	0	0

Tabla A.2: Valores de Preferencias Elementales obtenidas al aplicar los Criterios Elementales sobre los valores de las métricas extraídas (ver Tabla A.1)

Nombre método/función	Cód. Fte Op. D- [a]	GUI⊕Cod Op. DA [b]	SUD⊕BC Op. DA [c]	Op. C+- [d]	IR Op.C-+ [e]
AddressBookGUI.AddressBookGUI.search	1	1	0.9	0.91	0.95
AddressBookGUI.AddressBookGUI.save	1	1	0.9	0.91	0.95
AddressBookGUI.AddressBookGUI.forward	0.88	0.96	0.9	0.91	0.93
AddressBookGUI.AddressBookGUI.back	0.88	0.96	0.9	0.91	0.93
AddressBookGUI.AddressBookGUI.delete	1	0.81	0.9	0.91	0.86
AddressBookGUI.AddressBookGUI.close	1	0.76	0.9	0.91	0.83
AddressBookGUI.AddressBookGUI._load_contacts	0.94	0.71	0.91	0.92	0.81
AddressBookGUI.AddressBookGUI.__init__	0.78	0.68	0.9	0.91	0.78
AddressBookGUI.AddressBookGUI.contains_contact	0.88	0.67	0.78	0.8	0.73
AddressBookGUI.AddressBookGUI.search_similar_contacts	0.88	0.67	0.5	0.52	0.59
AddressBookGUI.Contact.get_id	0.56	0.42	0.7	0	0
AddressBookGUI.Contact.__init__	0.58	0.44	0.5	0	0
AddressBookGUI.Contact.get_address	0.56	0.42	0.09	0	0
AddressBookGUI.Contact.get_phone	0.56	0.42	0.09	0	0
AddressBookGUI.AddressBookGUI.clear	0.76	0.94	0	0	0
AddressBookGUI.Contact.__eq__	0.86	0.65	0	0	0
AddressBookGUI.AddressBookGUI.info_dialog	0.73	0.56	0	0	0
AddressBookGUI.Contact.get_name	0.56	0.42	0	0	0
AddressBookGUI.Contact.set_address	0.56	0.42	0	0	0
AddressBookGUI.Contact.set_name	0.56	0.42	0	0	0
AddressBookGUI.Contact.set_id	0.56	0.42	0	0	0
AddressBookGUI.Contact.set_phone	0.56	0.42	0	0	0

Tabla A.3: Preferencias Parciales y Globales (valores de IR) para el ejemplo libreta de contactos obtenidas al resolver la Estructura de Agregación para cada nodo. La estructura toma como base las Preferencias Elementales de la Tabla A.2)

[a] Agrega las Pref. Elementales de *Métricas de Código Fuente* [b] Agrega la Pref. Elemental de *Peso GUI* con la Pref. Parcial de *Métricas de código fuente* [c] Agrega las Pref. Elementales de *SUD* e *Intermediación (BC)* [d] Agrega la Pref. Parcial resultante de SUD⊕BC con la Pref. Elemental de *DI* [e] Computa la Preferencia Global (representa la metrica IR)

Detalles de Implementación ya que son métodos de acceso (por ejemplo `get_phone` y `set_address`), funciones integradas (como es el caso de `__eq__`) o constructores de clases que representan entidades del Dominio (por ejemplo el método `__init__` de `Contacto`). Por otra parte también se pueden visualizar métodos que obtuvieron cero para IR debido a que sus valores de *Intermediación* y *Utilidad Estructural* también lo son. Como se explicó en apartados anteriores, este factor hace que todos estos métodos obtengan un valor de cero para la métrica IR.

A.2.3. Filtrado de Nodos en el GELMF

En la Sección 6.5 se presenta un algoritmo que describe de manera concisa el proceso de filtrado que propone SAPSI. El mismo posee una opción para definir si el filtrado se realiza de manera iterativa o en una sola pasada. El principal criterio de selección de esta opción es el tamaño del sistema y la cantidad de información que se desee filtrar. Para el caso del ejemplo tratado en este anexo, el filtrado de una sola pasada es el más conveniente ya que el sistema es muy pequeño.

Para poder proceder, el algoritmo requiere de un valor umbral para determinar qué métodos/funciones serán filtrados por SAPSI. La determinación de dicho valor requiere de un análisis que considera diversos aspectos tales como el volumen de información que se desea filtrar, el tamaño del sistema bajo análisis, el tipo de sistema, los paradigmas de programación utilizados, etc.

Establecer un valor de umbral alto puede decantar en el filtrado de elementos que reflejan algunos aspectos relevantes del sistema. Por otra parte, si se determina un valor muy bajo puede que no se filtren los elementos necesarios para facilitar el análisis llevado a cabo por la estrategia.

Para el caso de la aplicación analizada en este anexo, establecer un valor de umbral es una tarea trivial ya que al ser un sistema pequeño, se pueden observar los métodos irrelevantes simplemente observando el GELMF y ciertos aspectos relativos a los nodos, como por ejemplo los nombres y las dependencias con otros. Uno de los aspectos que resaltan de esta observación es que aproximadamente la mitad de los nodos del GELMF obtienen cero para IR, lo cual indica que son utilidades del sistema. La mayoría de estos

son detalles de implementación y es por este motivo que obtienen dichos valores, aunque existen algunos ejemplos como `AddressBook.info_dialog` o `AddressBook.clear` que no caen dentro de este grupo aunque el valor para la métrica IR sea igualmente cero. Teniendo en cuenta este tipo de casos, se puede ver que el primero posee un alto valor de *SUD*, esto es porque es invocado numerosas veces desde distintos nodos. Además, dicho método no es un método importante respecto a la topología del GELMF, lo cual es un fuerte indicio que es un método irrelevante para el sistema. Analizando la función dentro de la aplicación, es posible determinar que dicho método no es determinante para el funcionamiento del sistema, de hecho, realizando algunos cambios podría ser eliminado sin afectar la estructura interna que determina el funcionamiento de la aplicación.

Por otra parte, el método `AddressBook.clear` posee las mismas características que el anterior, sin embargo este es el manejador de una funcionalidad que brinda el sistema la cual es limpiar los campos de texto del formulario principal de la GUI. En este caso el elemento no es filtrado ya que los análisis posteriores toman como punto de partida este tipo de nodos en función de que los mismos están estrechamente vinculados con las acciones que puede realizar el usuario con el sistema.

Al analizar la Tabla A.3 para este caso en particular, es posible comprobar que la mayoría de los métodos con valor de IR distinto de cero obtuvieron un valor alto para esta métrica. Esto se debe al acotado tamaño del GELMF y la cantidad de detalles de implementación que posee el sistema. Por lo tanto, interpretando la importancia en términos de utilidad (es decir que los nodos con valores de IR bajos deben ser eliminados), el resultado del algoritmo de filtrado es invariante para umbrales en el intervalo [0,0.58]. Esto indica que para filtrar más nodos se debe seleccionar un umbral más alto y el resultado de esto puede decantar en la eliminación de nodos con grados de utilidad que pueden no representar elementos irrelevantes. Por ejemplo, para la aplicación analizada, si se eleva el umbral a 0.59, se filtra el método `AddressBook.search_similar_contacts` el cual permite buscar contactos que contengan el término de la consulta. Si bien no es un elemento esencial del funcionamiento del sistema, dicho método es dependencia de dos funcionalidades principales del mismo: búsqueda (`AddressBook.search`) y eliminación (`AddressBook.delete`).

Por otra parte, si se mantiene el mismo umbral y se elige el filtrado iterativo, el

resultado de calcular nuevamente las métricas en cada iteración resulta en la eliminación de todos los nodos que no son "manejadores" ni "inicializadores". Es por este motivo que la opción de filtrado iterativo se debe utilizar para sistemas de tamaño grande y cuando las pruebas determine que el filtrado común no reduce la densidad del GELMF lo suficiente para proceder con los análisis posteriores.

Con respecto a la estimación de umbrales en los procesos de evaluación de software, como bien han analizado distintos autores en el contexto de Ingeniería de Software (Erdogmus y Tanir, 2002; Mathias y cols., 1999; Hamou-Lhadj y Lethbridge, 2006; Steidl y cols., 2012), dicha tarea conlleva el estudio de ciertos factores como la inspección de los objetos bajo estudio, la experiencia de un grupo de expertos en el entorno en conjunto con la realización de pruebas y el análisis de los resultados obtenidos. Para el caso de las aplicaciones inspeccionadas en los casos de estudio de esta tesis, se tomaron en cuenta todos los criterios previamente mencionados. Estos casos de estudio presentan un GELMF con otras características y tamaños donde el espectro de valores de IR obtenidos es más amplio y variado. En estos casos, la definición de un umbral siguiendo los lineamientos antes mencionados es más sustancial.

A.3. Técnica de Clustering y Generación del Modelo de Casos de Uso

La próxima etapa en la estrategia recibe como entrada el GELMF filtrado y procede a la ejecución del algoritmo de clustering para obtener el modelo de cluster del sistema analizado. Posteriormente, se debe interactuar con la herramienta para definir los rótulos en el modelo y una vez completada esta tarea se procede a construir el modelo de casos de uso que será generado en un archivo con formato XMI.

A.3.1. Modelo de Cluster

En la Figura A.5 se muestran los nodos que pasaron el proceso de filtrado descripto en el Capítulo 6. además, en la misma figura se ve reflejado el resultado de aplicar el algoritmo de clustering explicado en el Capítulo 7. Esto se puede visualizar claramente

en esta primera vista donde los nodos que poseen el mismo color han sido agrupados en el mismo cluster.

Salvo el cluster que agrupa los nodos iniciales, todos los demás corresponden a clusters manejadores[5]. Debido a que el GELMF filtrado no está compuesto por muchos nodos y la mayoría son manejadores, el modelo de clusters no contiene grupos que hayan sido generados por reglas de agrupamiento estructural ni fusiones entre clusters (ver Sección 7.2.3).

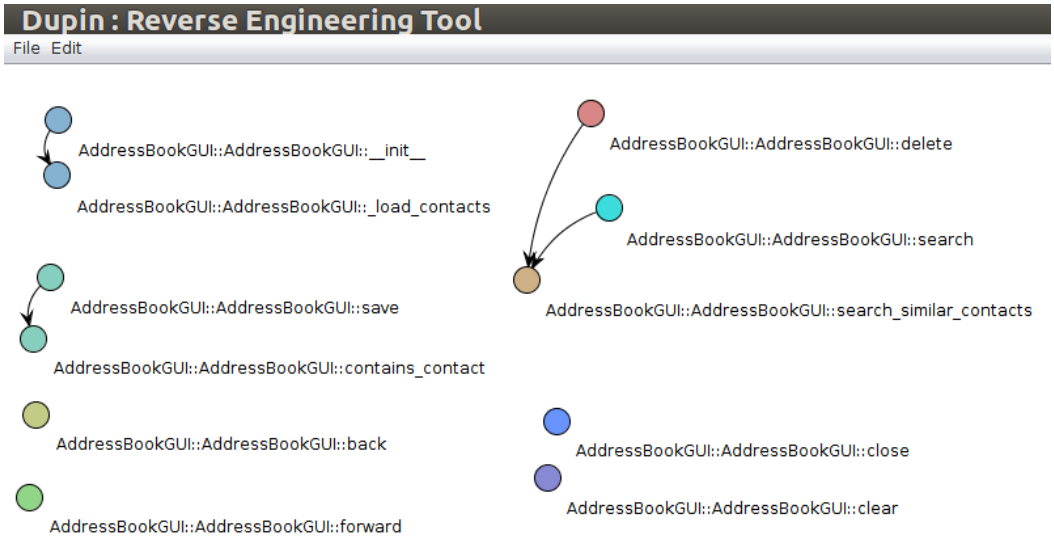

Figura A.5: Visualización del GELMF filtrado y agrupado de acuerdo al proceso propuesto por SAPSI para la aplicación libreta de contactos. La visualización corresponde a la herramienta *Dupin*.

En la Figura A.6 se muestra una captura completa de la aplicación donde se visualizan el modelo de clusters correspondiente a la Figura A.5. En la herramienta *Dupin* esto se logra presionando el botón `Collapse Clusters` que agrupa los nodos de cada cluster representándolos con un nodo elíptico. En esta figura se puede observar una captura de la herramienta completa, donde en la parte inferior se visualiza el panel con información de cada cluster seleccionado.

Como es posible apreciar, los diferentes clusters se encuentran nombrados con una precisión considerable, al punto en que es posible determinar qué hace cada funcionalidad. Esto representa un escenario favorable para el ingeniero de software ya que no deberá

[5]Un cluster que fue generado partiendo de un método/función manejador.

realizar grandes alternaciones a los rótulos definidos de manera automática. En ciertas ocasiones, algunos elementos del modelo pueden obligar a llevar a cabo una inspección más profunda al modelo de clusters en busca de los términos más correctos para rotular cada cluster. Es importante aclarar que estos casos donde se debe analizar con mayor detenimiento el modelo suelen darse en sistemas de mayor envergadura. Es decir, por lo general en estos casos muchos clusters son nombrados de manera automática, sin embargo el conjunto de clusters que no tienen un rótulo representativo definido mediante esta forma suele ser la mayoría.

Para el caso del ejemplo analizado, los nombres extraídos de manera automática coinciden con el texto de los botones de la GUI o en su defecto con el nombre del manejador. Si bien estos son bastante claros, se puede mejorar la comprensión de los clusters del ejemplo extendiendo los nombres para dejar en claro la semántica de cada funcionalidad. Algunos ejemplos podrían ser:

- para el cluster `save` se puede establecer el rótulo *Save Contact* de acuerdo a la información provista por el tooltip del componente GUI interactivo.

- `Save & Close` se puede renombrar como *Save Address Book & Exit*.

- el cluster `clear` se puede rotular como *Clear Form*.

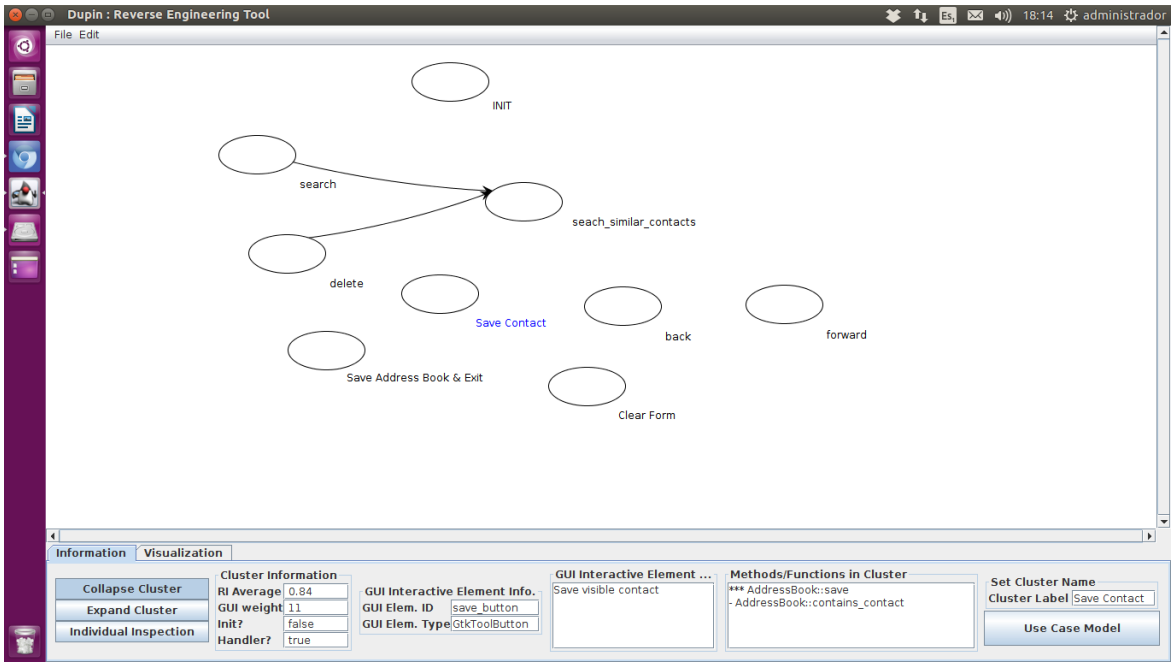

Figura A.6: Visualización del modelo de cluster para la aplicación libreta de contactos. La visualización corresponde a la herramienta *Dupin*.

A.3.2. Modelo de Casos de Uso UML

Finalmente, una vez que se han verificado todos los nombres de los clusters es posible generar el modelo de casos de uso UML presionando el botón *Use Case Model* de la interfaz de la herramienta *Dupin*. Esto genera un archivo XMI en el directorio raíz del proyecto que puede ser examinado en cualquier herramienta de UML que siga el estándar de la *Object Management Group* (OMG). Mediante este tipo de herramientas también es posible consultar los métodos/funciones que implementan la funcionalidad de cada de uso, dicha información se incorpora como comentarios dentro de cada caso de uso.

En la Figura A.7 se muestra el modelo de casos de uso visualizado con la herramienta Modelio (Modelio, 2016) a partir del archivo XMI generado por la herramienta. El contenido del archivo XMI también es anexado luego de la imagen para su verificación[6].

[6]Para la correcta disposición del texto contenido en el archivo XMI, se han insertado caracteres de "nueva línea", los mismos deben ser eliminados para facilitar la importación del archivo en Modelio

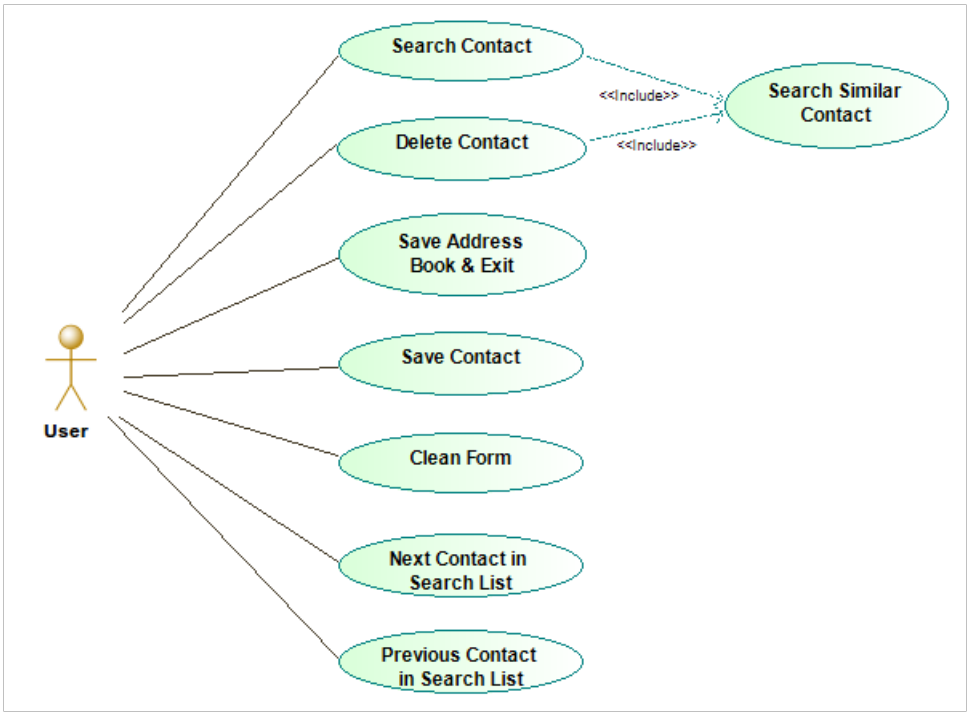

Figura A.7: Visualización del modelo de casos de uso para la aplicación libreta de contactos. La visualización corresponde a la herramienta Modelio.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmlns:uml="http://www.omg.org/spec/UML/20110701"
        xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmi:version="2.1"
        xmi:id="SAPSI_OUTPUT" name="ucmodel">
    <packagedElement xmi:type="uml:Actor"
            xmi:id="User" name="User"/>
    <packagedElement xmi:type="uml:Association" xmi:id="User-Save_Address_Book_Exit"
            memberEnd="User-Save_Address_Book_Exit_end User-Save_Address_Book_Exit_begin"
            navigableOwnedEnd="User-Save_Address_Book_Exit_end">
        <ownedEnd xmi:type="uml:Property"
                xmi:id="User-Save_Address_Book_Exit_end"
                visibility="public" type="Save_Address_Book_Exit"
                association="User-Save_Address_Book_Exit"/>
        <ownedEnd xmi:type="uml:Property"
                xmi:id="User-Save_Address_Book_Exit_begin"
                visibility="public" type="User"
                association="User-Save_Address_Book_Exit"/>
    </packagedElement>
    <packagedElement xmi:type="uml:Association"
            xmi:id="User-Save_Contact"
            memberEnd="User-Save_Contact_end User-Save_Contact_begin"
            navigableOwnedEnd="User-Save_Contact_end">
        <ownedEnd xmi:type="uml:Property"
                xmi:id="User-Save_Contact_end" v
                isibility="public" type="Save_Contact"
                association="User-Save_Contact"/>
        <ownedEnd xmi:type="uml:Property"
                xmi:id="User-Save_Contact_begin"
                visibility="public" type="User"
                association="User-Save_Contact"/>
    </packagedElement>
    <packagedElement xmi:type="uml:Association"
            xmi:id="User-Clean_Form"
            memberEnd="User-Clean_Form_end User-Clean_Form_begin"
            navigableOwnedEnd="User-Clean_Form_end">
        <ownedEnd xmi:type="uml:Property"
                xmi:id="User-Clean_Form_end"
                visibility="public" type="Clean_Form"
                association="User-Clean_Form"/>
```

```
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Clean_Form_begin"
            visibility="public" type="User"
            association="User-Clean_Form"/>
</packagedElement>
<packagedElement xmi:type="uml:Association"
        xmi:id="User-Next_Contact_in_Search_List"
        memberEnd="User-Next_Contact_in_Search_List_end User-Next_Contact_in_Search_List_begin"
        navigableOwnedEnd="User-Next_Contact_in_Search_List_end">
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Next_Contact_in_Search_List_end"
            visibility="public" type="Next_Contact_in_Search_List"
            association="User-Next_Contact_in_Search_List"/>
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Next_Contact_in_Search_List_begin"
            visibility="public" type="User"
            association="User-Next_Contact_in_Search_List"/>
</packagedElement>
<packagedElement xmi:type="uml:Association"
        xmi:id="User-Previous_Contact_in_Search_List"
        memberEnd="User-Previous_Contact_in_Search_List_end User-Previous_Contact_in_Search_List_begin"
        navigableOwnedEnd="User-Previous_Contact_in_Search_List_end">
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Previous_Contact_in_Search_List_end"
            visibility="public" type="Previous_Contact_in_Search_List"
            association="User-Previous_Contact_in_Search_List"/>
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Previous_Contact_in_Search_List_begin" visibility="public" type="User"
            association="User-Previous_Contact_in_Search_List"/>
</packagedElement>
<packagedElement xmi:type="uml:Association"
        xmi:id="User-Search_Contact"
        memberEnd="User-Search_Contact_end User-Search_Contact_begin"
        navigableOwnedEnd="User-Search_Contact_end">
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Search_Contact_end"
            visibility="public" type="Search_Contact"
            association="User-Search_Contact"/>
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Search_Contact_begin"
            visibility="public" type="User"
            association="User-Search_Contact"/>
</packagedElement>
<packagedElement xmi:type="uml:Association"
        xmi:id="User-Delete_Contact"
        memberEnd="User-Delete_Contact_end User-Delete_Contact_begin"
        navigableOwnedEnd="User-Delete_Contact_end">
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Delete_Contact_end"
            visibility="public" type="Delete_Contact"
            association="User-Delete_Contact"/>
    <ownedEnd xmi:type="uml:Property"
            xmi:id="User-Delete_Contact_begin"
            visibility="public" type="User"
            association="User-Delete_Contact"/>
</packagedElement>
<packagedElement xmi:type="uml:UseCase"
        xmi:id="Search_Contact"
        name="Search Contact">
    <ownedComment xmi:type="uml:Comment"
            xmi:id="Search_Contact_Elements">
        <body>AddressBookGUI::AddressBookGUI::search</body>
    </ownedComment>
    <include xmi:type="uml:Include"
            xmi:id="Search_Contact-Search_Similar_Contact"
            name="UseCaseDependency" addition="Search_Similar_Contact"/>
</packagedElement>
<packagedElement xmi:type="uml:UseCase"
        xmi:id="Delete_Contact"
        name="Delete Contact">
    <ownedComment xmi:type="uml:Comment"
            xmi:id="Delete_Contact_Elements">
        <body>AddressBookGUI::AddressBookGUI::delete</body>
    </ownedComment>
    <include xmi:type="uml:Include"
            xmi:id="Delete_Contact-Search_Similar_Contact"
            name="UseCaseDependency" addition="Search_Similar_Contact"/>
```

```
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Save_Address_Book_Exit"
            name="Save Address Book &amp;amp; Exit">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Save_Address_Book_Exit_Elements">
            <body>AddressBookGUI::AddressBookGUI::close</body>
        </ownedComment>
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Save_Contact"
            name="Save Contact">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Save_Contact_Elements">
            <body>AddressBookGUI::AddressBookGUI::save
                AddressBookGUI::AddressBookGUI::contains_contact</body>
        </ownedComment>
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Clean_Form"
            name="Clean Form">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Clean_Form_Elements">
            <body>AddressBookGUI::AddressBookGUI::clear</body>
        </ownedComment>
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Next_Contact_in_Search_List"
            name="Next Contact in Search List">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Next_Contact_in_Search_List_Elements">
            <body>AddressBookGUI::AddressBookGUI::forward</body>
        </ownedComment>
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Previous_Contact_in_Search_List"
            name="Previous Contact in Search List">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Previous_Contact_in_Search_List_Elements">
            <body>AddressBookGUI::AddressBookGUI::back</body>
        </ownedComment>
    </packagedElement>
    <packagedElement xmi:type="uml:UseCase"
            xmi:id="Search_Similar_Contact"
            name="Search Similar Contact">
        <ownedComment xmi:type="uml:Comment"
                xmi:id="Search_Similar_Contact_Elements">
            <body>AddressBookGUI::AddressBookGUI::search_similar_contacts</body>
        </ownedComment>
    </packagedElement>
</uml:Model>
```

Anexo B

Funciones de Conjunción-Disyunción Generalizada

¿Cómo puede ser que las Matemáticas, un producto del pensamiento humano independiente de la experiencia, se adapte tan admirablemente a objetos de la realidad?

—Albert Einstein

En la Tabla B.1 se exhibe una lista con 20 funciones de Conjunción-Disyunción Generalizada y sus respectivos símbolos y valores correspondientes a r y d.

Operador	Símbolo	Grado Dis.	r			
			n=2	n=3	n=4	n=5
Disyunción Pura	D	1.0000	$+\infty$	$+\infty$	$+\infty$	$+\infty$
QD Fuerte(+)	$D++$	0.9375	20.63	24.3	27.71	30.09
QD Fuerte	$D+$	0.8750	9.52	11.09	12.27	13.235
QD Fuerte(-)	$D+-$	0.8125	5.8	6.68	7.32	7.82
QD media	DA	0.7500	3.93	4.45	4.82	5.11
QD Débil(+)	$D-+$	0.6875	2.79	3.1	3.32	3.48
QD Débil	$D-$	0.6250	2.02	2.19	2.3	2.38
Media Cuadrática	SQU	0.6232	2			
QD Débil(-)	$D--$	0.5625	1.45	1.52	1.57	1.6
Media Aritmética	A	0.5000	1.00	1.00	1.00	1.00
QC Débil(-)	$C--$	0.4375	0.62	0.57	0.55	5.26
QC Débil	$C-$	0.3750	0.26	0.19	0.15	0.13
Media Geométrica	GEO	0.333	0			
QC Débil(+)	$C-+$	0.3125	-0.15	-0.21	-0.24	-0.25
QC media	CA	0.2500	-0.72	-0.73	-0.72	-0.70
Media Armónica	HAR	0.2274	-1			
QC Fuerte(-)	$C+-$	0.1875	-1.66	-1.55	-1.46	-1.38
QC Fuerte	$C+$	0.1250	-3.51	-3.11	-2.82	-2.6
QC Fuerte(+)	$C++$	0.0625	-9.06	-7.64	-6.69	-6.01
Conjunción Pura	C	0.0000	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Tabla B.1: Valores para r correspondientes a cada función Conjunción/Disyunción Generalizadas (CDG) y el número de operandos (n). La abreviación "QD" significa Cuasi Disyunción mientras que "QC" representa Cuasi Conjunción. La columna 'Grado Dis.' muestra el grado de "simultaneidad" del operador, es decir el grado de disyunción.

Anexo C

Modelos de Casos de Uso para los Sistemas Analizados

Toda la historia de la Ingeniería de Software está relacionada con el aumento en los niveles de abstracción.

—Grady Booch

En este anexo se muestran los modelos de casos de uso obtenidos para los sistemas presentados como casos de estudio en esta tesis doctoral. Por cada uno de los sistemas analizados se visualiza el modelo ideal y el obtenido con la estrategia.

Para el análisis y comparación de dichos modelos es necesario tener en cuenta los siguientes aspectos:

- Para facilitar la visualización de los modelos, cada uno de estos se exhibe mediante distintas imágenes donde se agrupan varias funcionalidades del sistema modelado.
- En un mismo modelo pueden existir ciertos casos de uso que se replican en las distintas imágenes, en estos casos el rótulo del caso de uso finaliza con un asterisco ("*") para denotar que dicho elemento del modelo figura en varias figuras.
- Los rótulos de aquellos casos de uso que representaban la misma funcionalidad en

ambos modelos, el ideal y el obtenido mediante la estrategia, se han unificado para mejorar la comparación entre los mismos.

C.1. Modelos para Febrl

C.1.1. Modelo Ideal (*gold standard*)

Funcionalidades Básicas

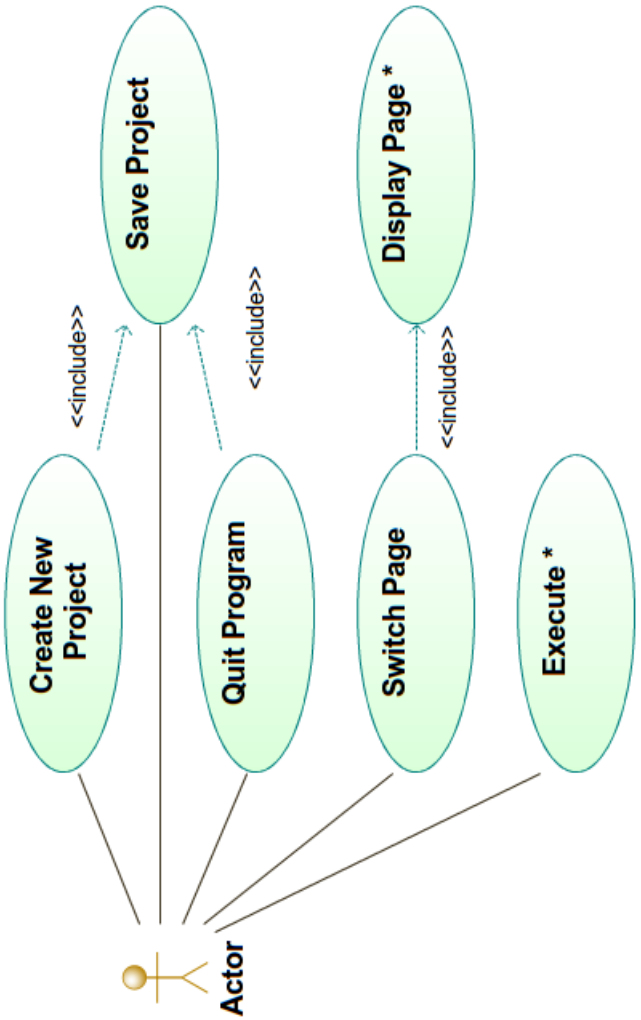

Figura C.1: Modelo ideal de casos de uso correspondiente a las funcionalidades básicas del sistema Febrl, tales como guardar, nuevo proyecto, etc.

Funcionalidades relacionadas con *Datos*

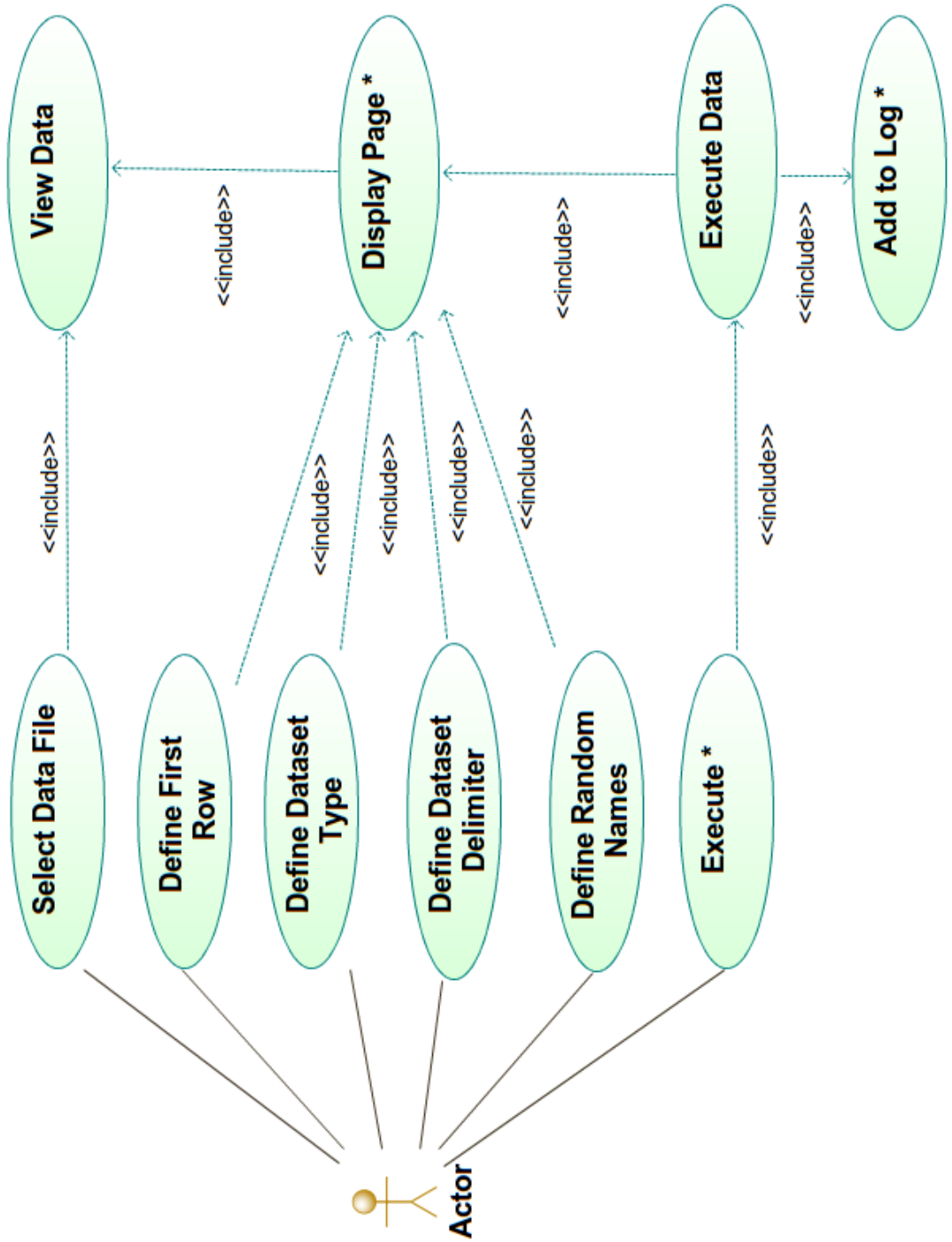

Figura C.2: Modelo ideal de casos de uso correspondiente a las funcionalidades para tratar distintos datos de base del sistema Febrl.

Funcionalidades *Explorar, Estandarizar e Indexar*

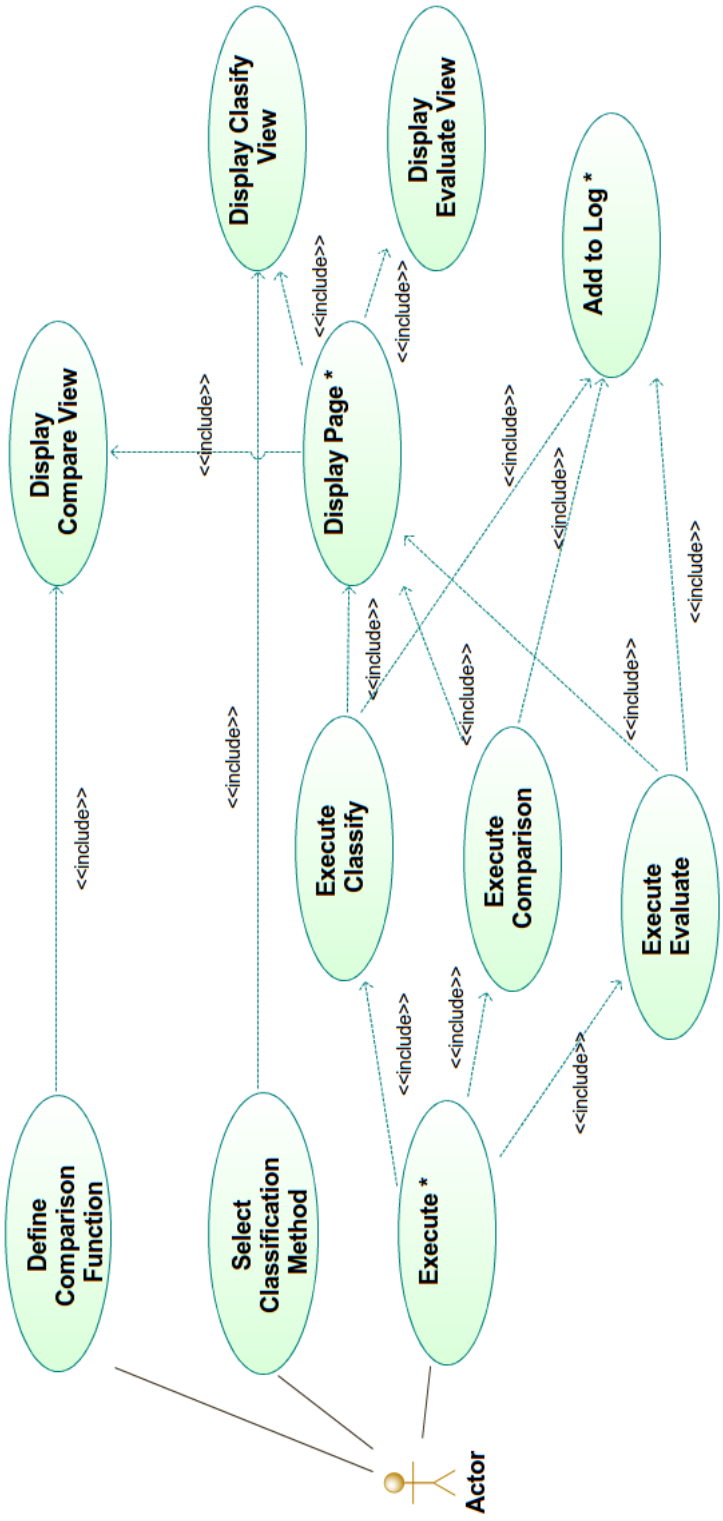

Figura C.3: Modelo ideal de casos de uso correspondiente a las funcionalidades *Explorar, Estandarizar e Indexar* del sistema Febrl.

Funcionalidades *Clasificar, Comparar y Evaluar*

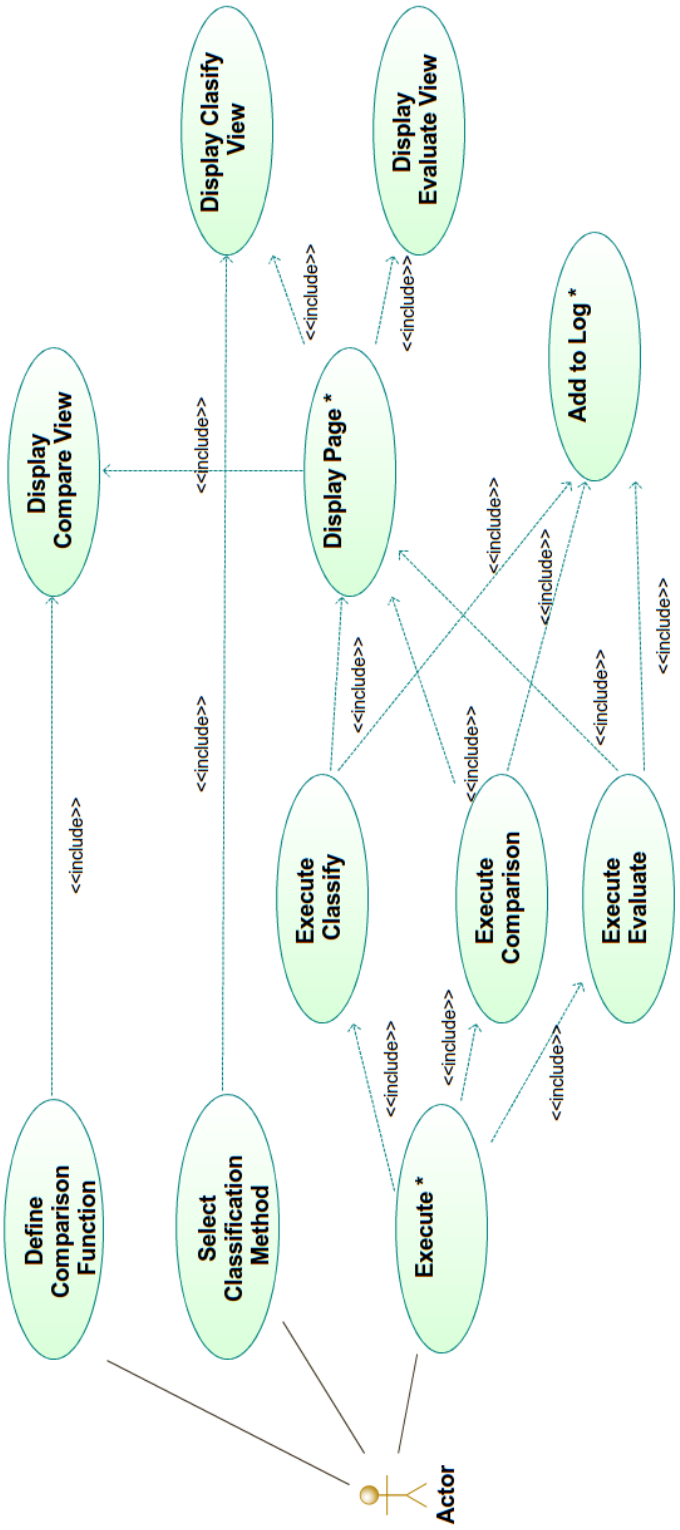

Figura C.4: Modelo ideal de casos de uso correspondiente a las funcionalidades *Clasificar, Comparar y Evaluar* del sistema Febrl.

Funcionalidad *Ejecutar*

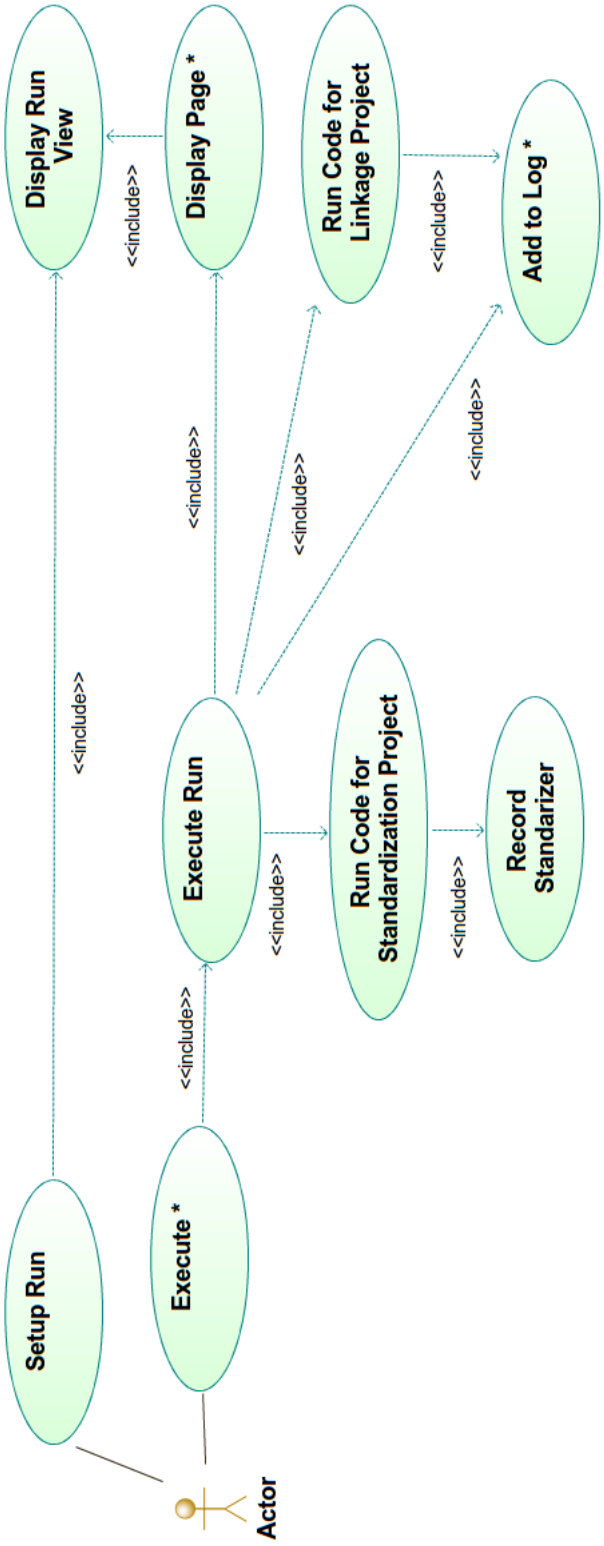

Figura C.5: Modelo ideal de casos de uso correspondiente a las funcionalidades relativas a *Ejecutar* del sistema Febrl.

C.1.2. Modelo Obtenido con SAPSI

Funcionalidades Básicas

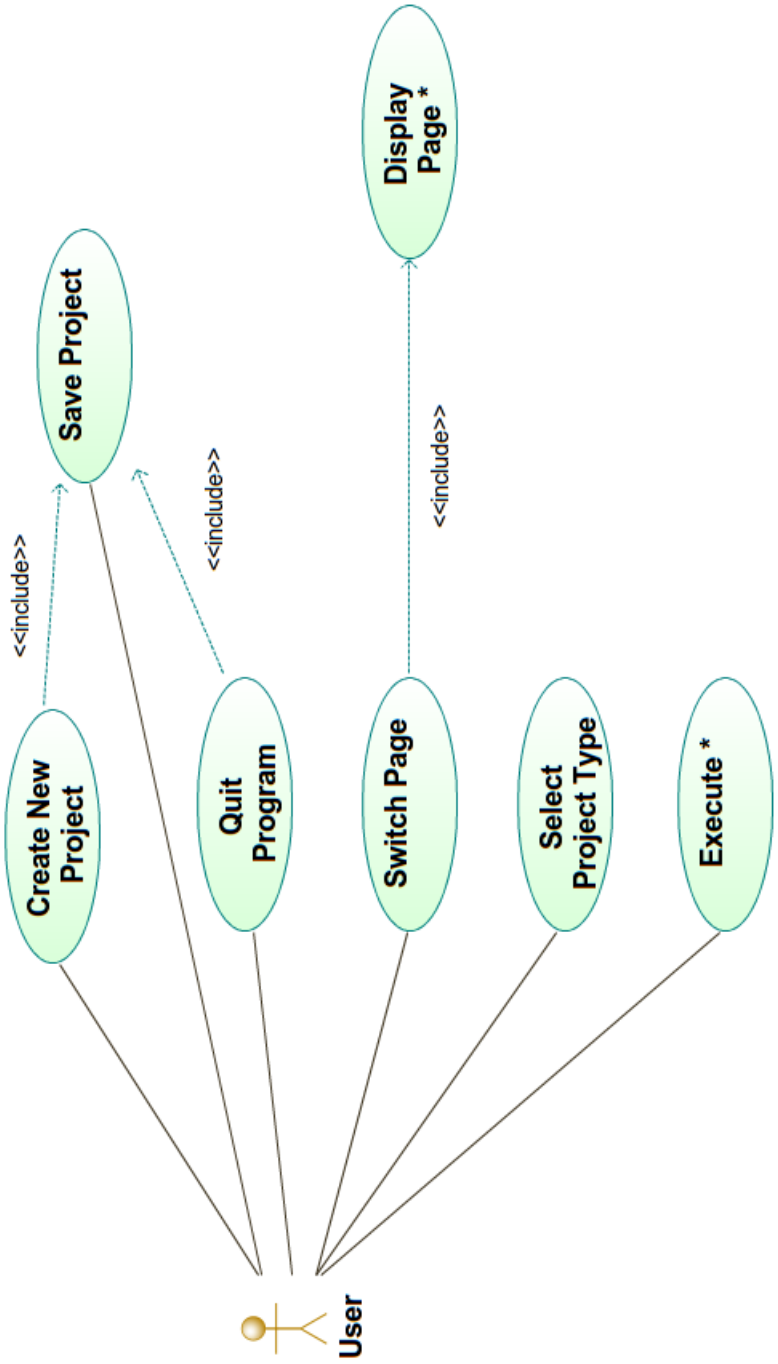

Figura C.6: Modelo SAPSI de casos de uso correspondiente a las funcionalidades básicas del sistema Febrl, tales como guardar, nuevo proyecto, etc.

Funcionalidades relacionadas con *Datos*

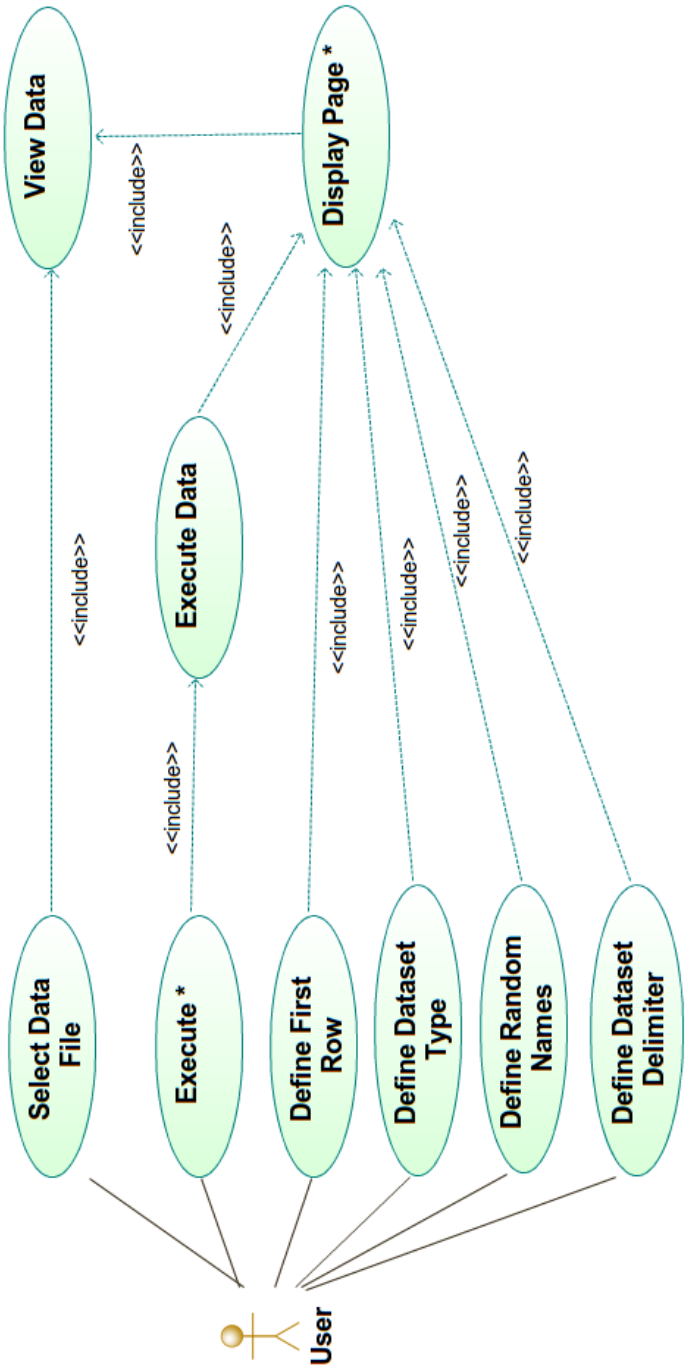

Figura C.7: Modelo SAPSI de casos de uso correspondiente a las funcionalidades para tratar distintos datos de base del sistema Febrl.

Funcionalidades *Explorar, Estandarizar e Indexar*

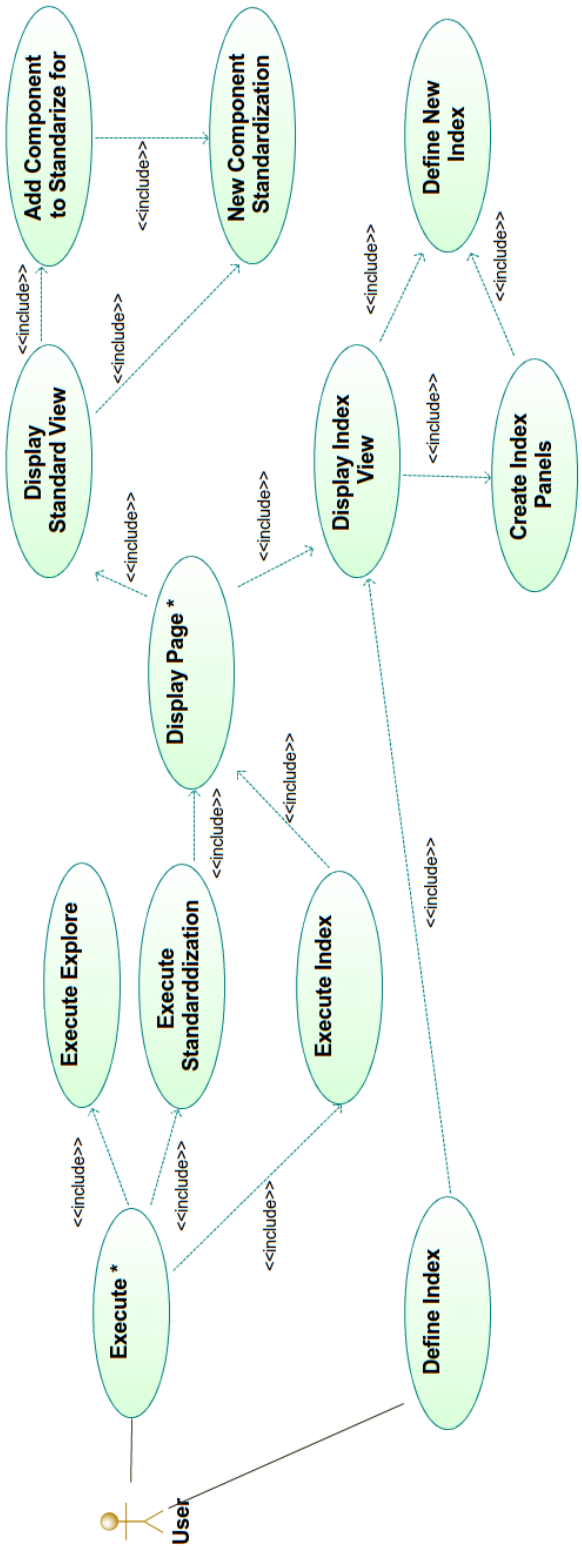

Figura C.8: Modelo SAPSI de casos de uso correspondiente a las funcionalidades *Explorar, Estandarizar e Indexar* del sistema Febrl.

Funcionalidades *Clasificar, Comparar y Evaluar*

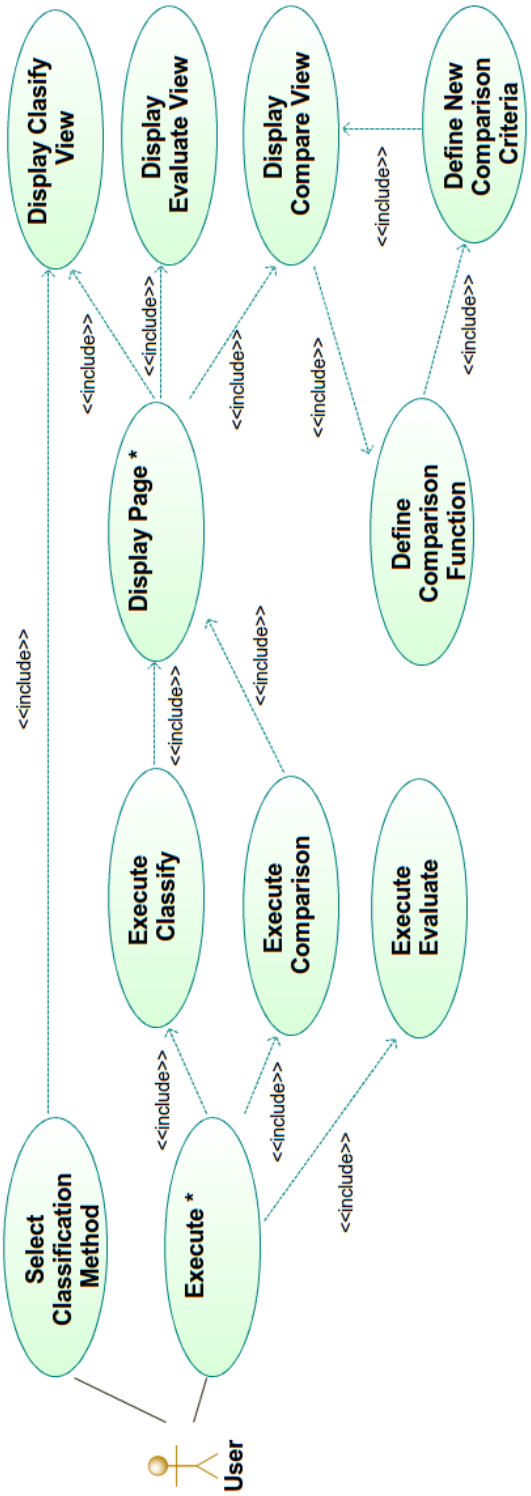

Figura C.9: Modelo SAPSI de casos de uso correspondiente a las funcionalidades *Clasificar, Comparar y Evaluar* del sistema Febrl.

Funcionalidad *Ejecutar*

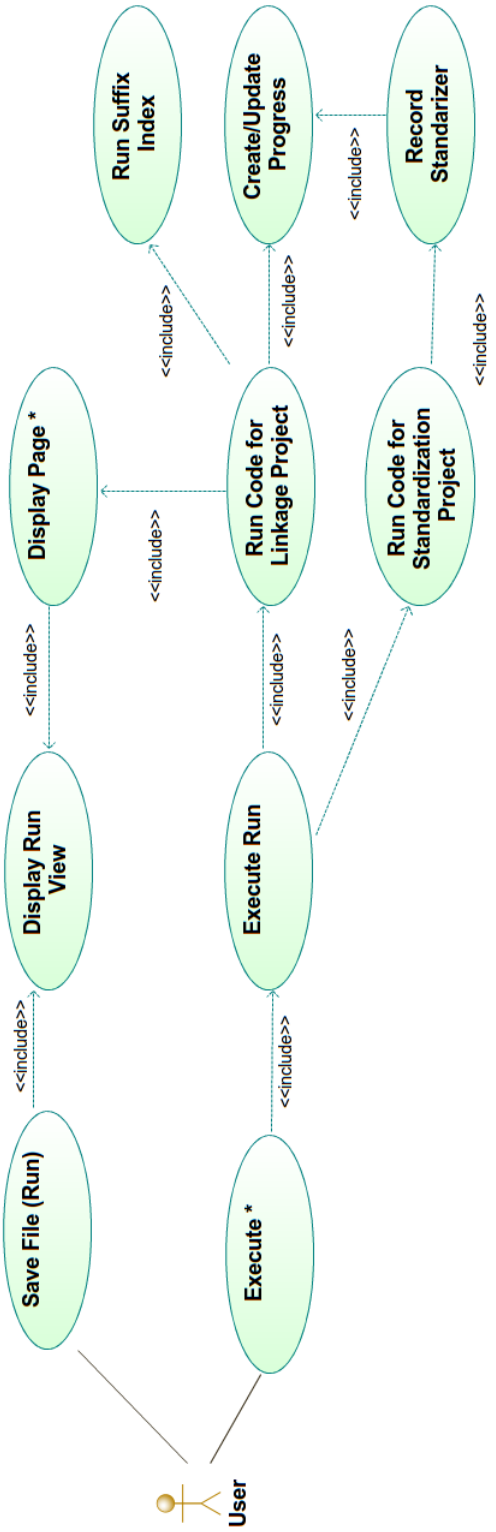

Figura C.10: Modelo SAPSI de casos de uso correspondiente a las funcionalidades relativas a *Ejecutar* del sistema Febrl.

C.2. Modelos para Pigeon Planner

C.2.1. Modelo Ideal (*gold standard*)

Barra de Menús

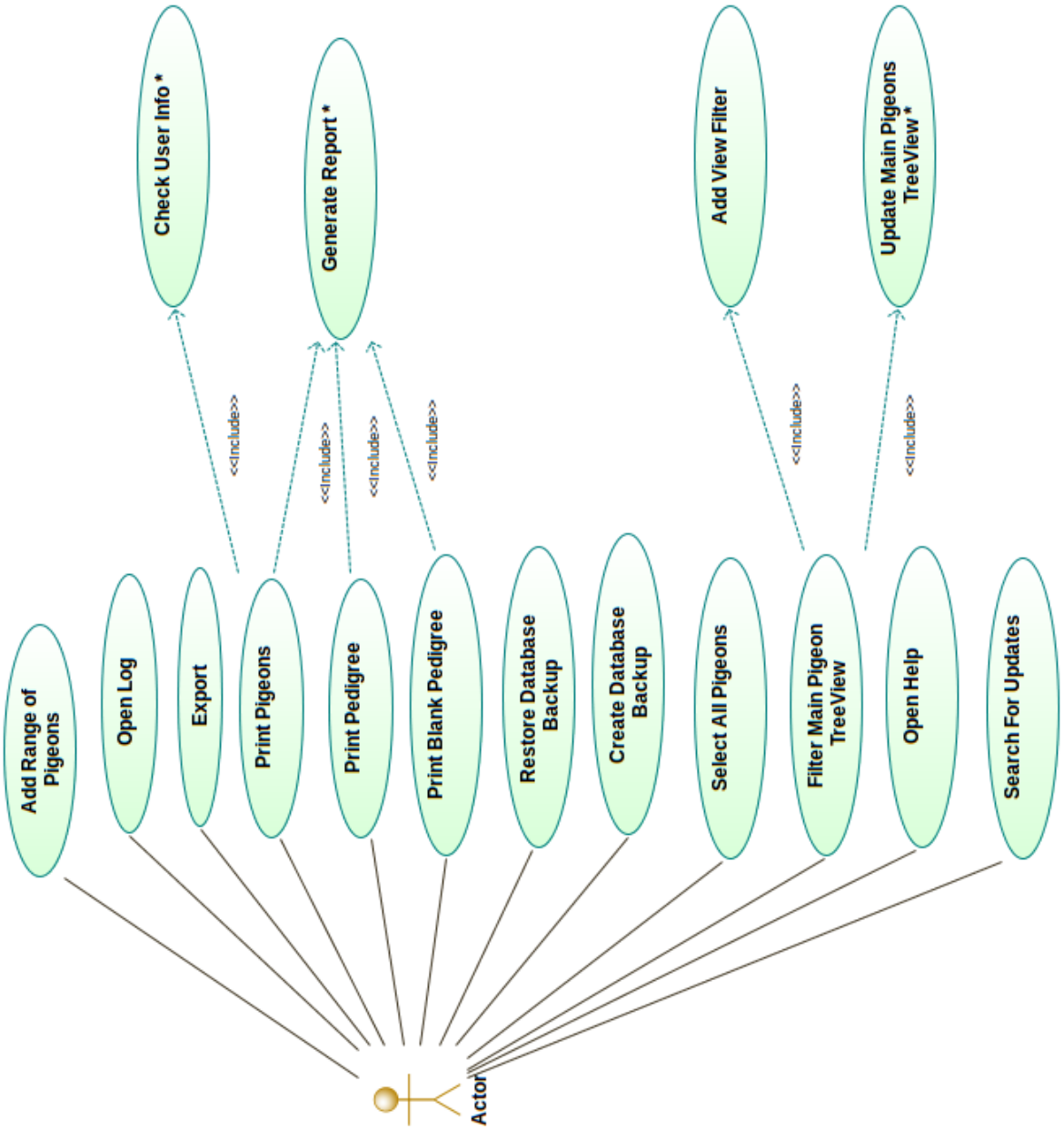

Figura C.11: Modelo ideal de casos de uso correspondiente a las funcionalidades de la barra de menús del sistema Pigeon Planner.

Barra de Herramientas

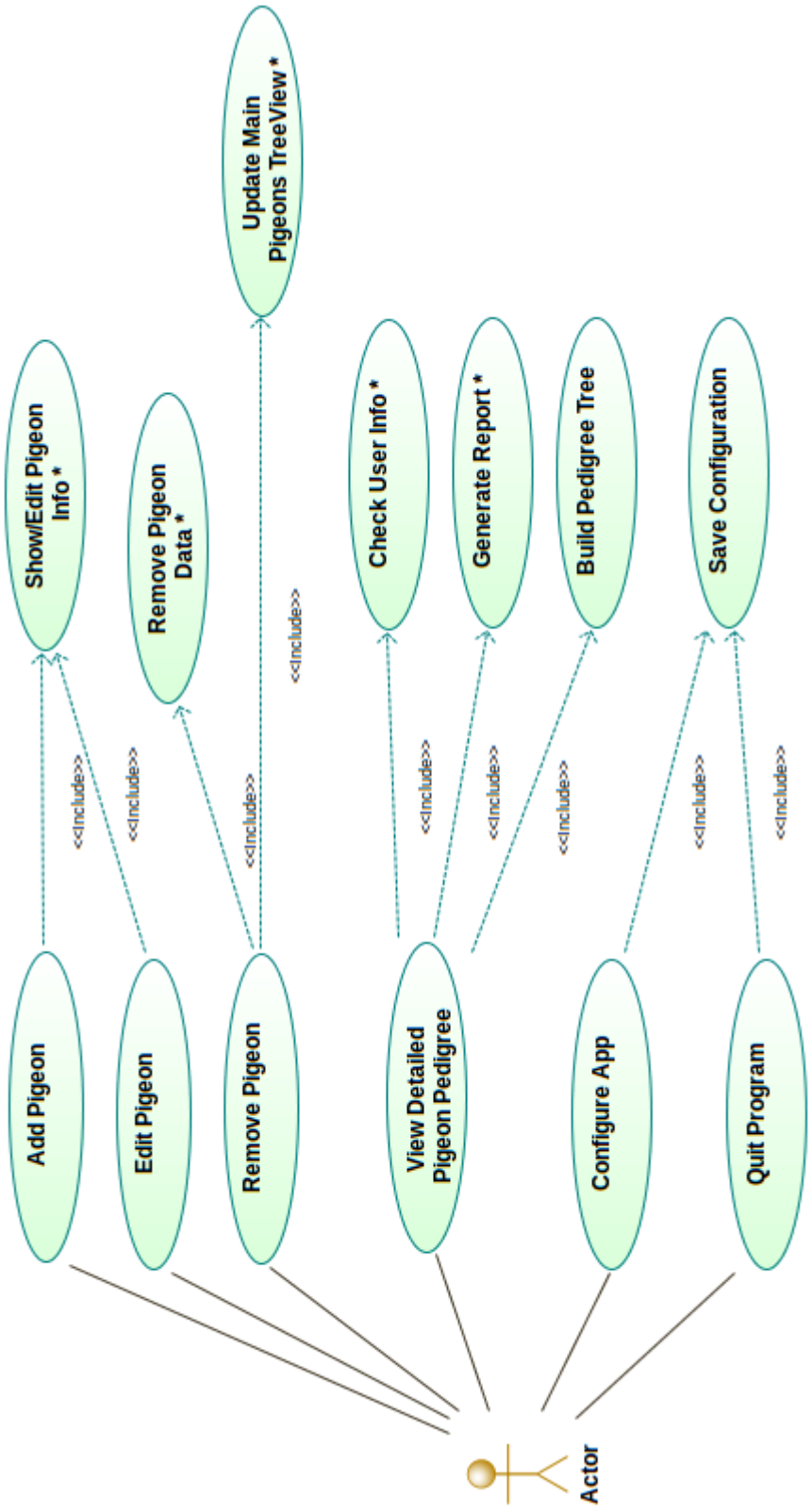

Figura C.12: Modelo ideal de casos de uso correspondiente a las funcionalidades de la barra de herramientas provistas por el sistema Pigeon Planner.

Herramientas

Figura C.13: Modelo ideal de casos de uso correspondiente a las funcionalidades de las herramientas provistas por el sistema Pigeon Planner.

Pestañas

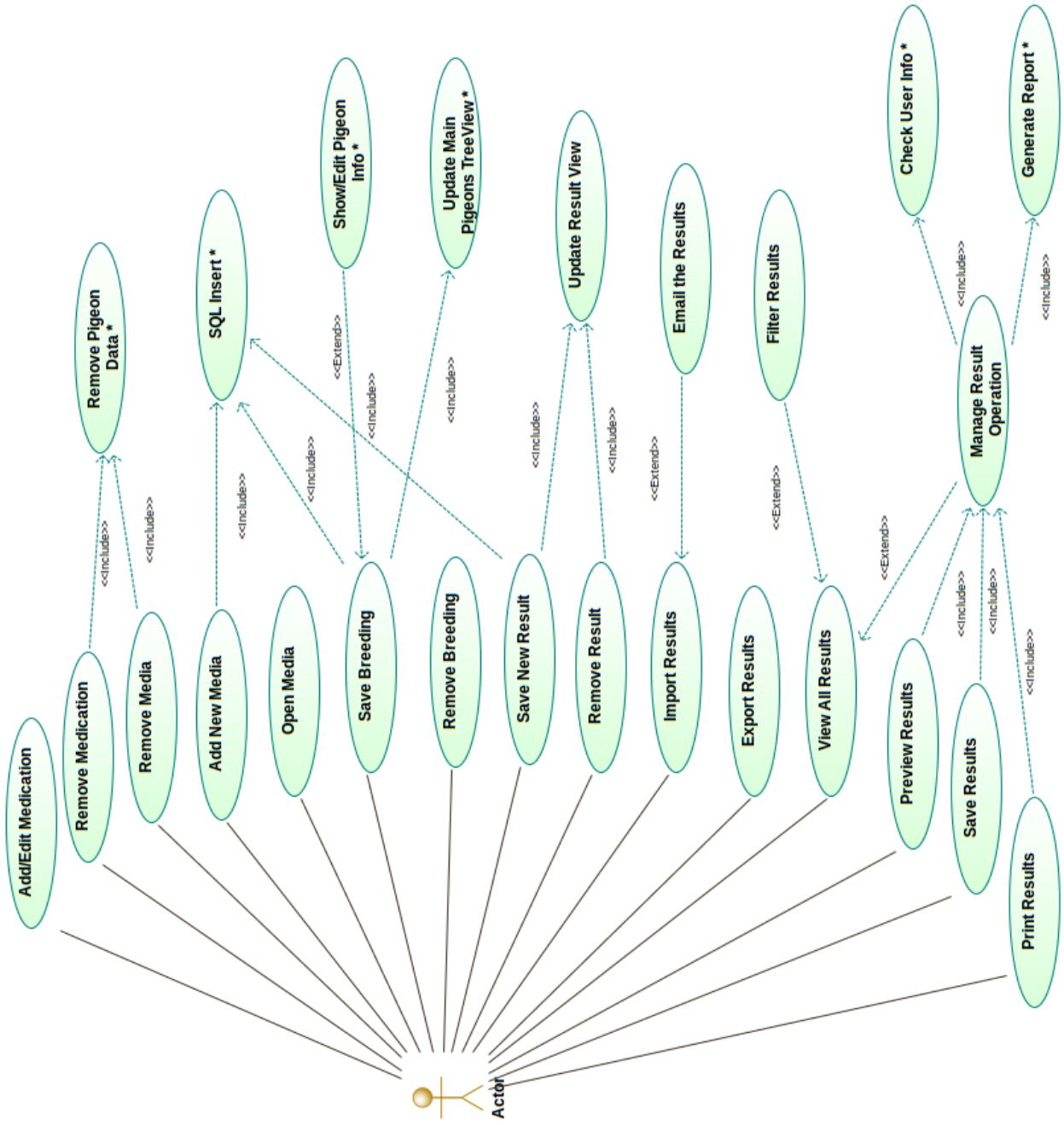

Figura C.14: Modelo ideal de casos de uso correspondiente a las funcionalidades encontradas en las pestañas del sistema Pigeon Planner.

C.2.2. Modelo Obtenido con SAPSI

Barra de Menús

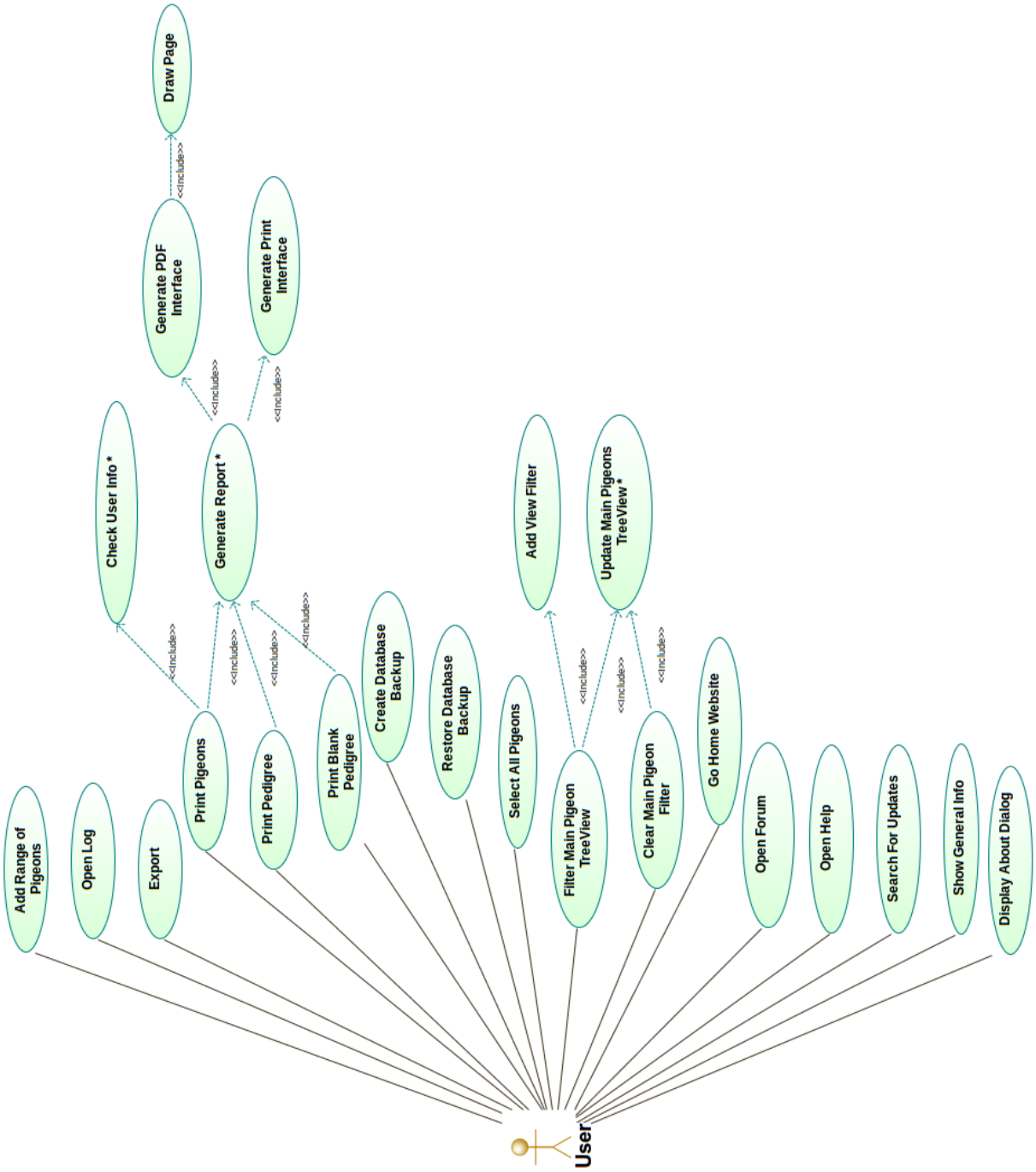

Figura C.15: Modelo obtenido con SAPSI correspondiente a las funcionalidades de la barra de menús del sistema Pigeon Planner.

Barra de Herramientas

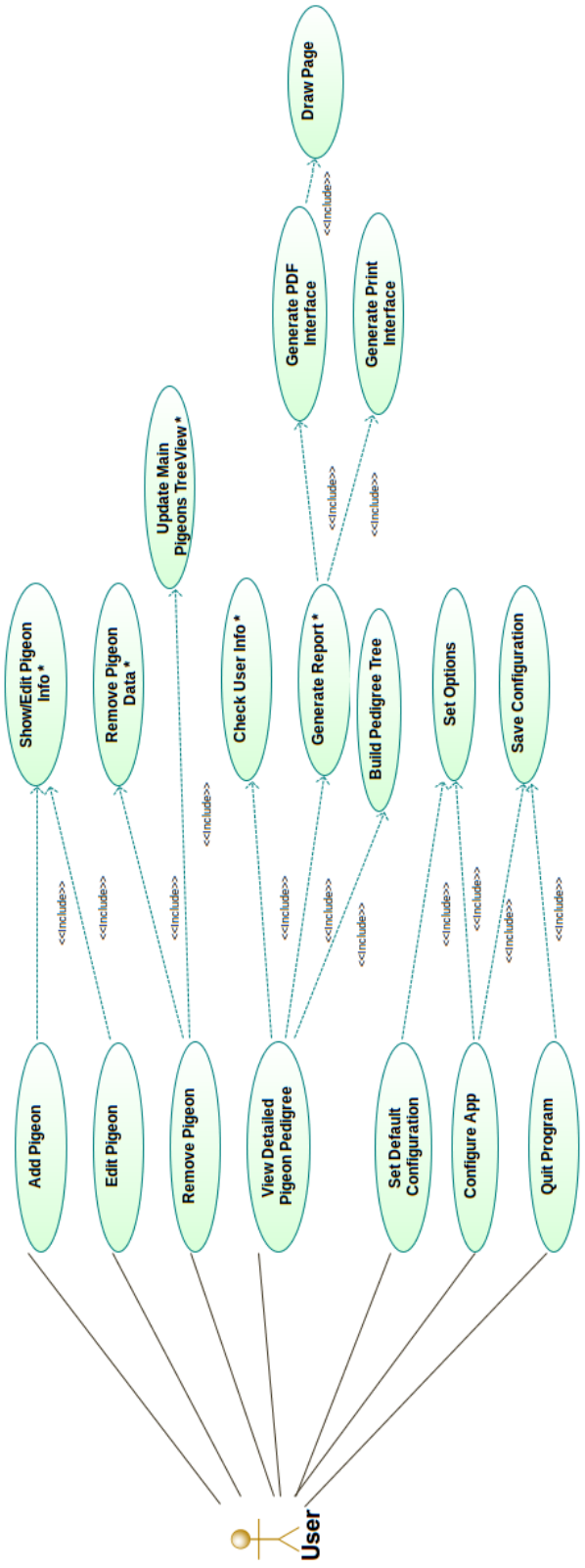

Figura C.16: Modelo obtenido con SAPSI correspondiente a las funcionalidades de la barra de herramientas provistas por el sistema Pigeon Planner.

Herramientas

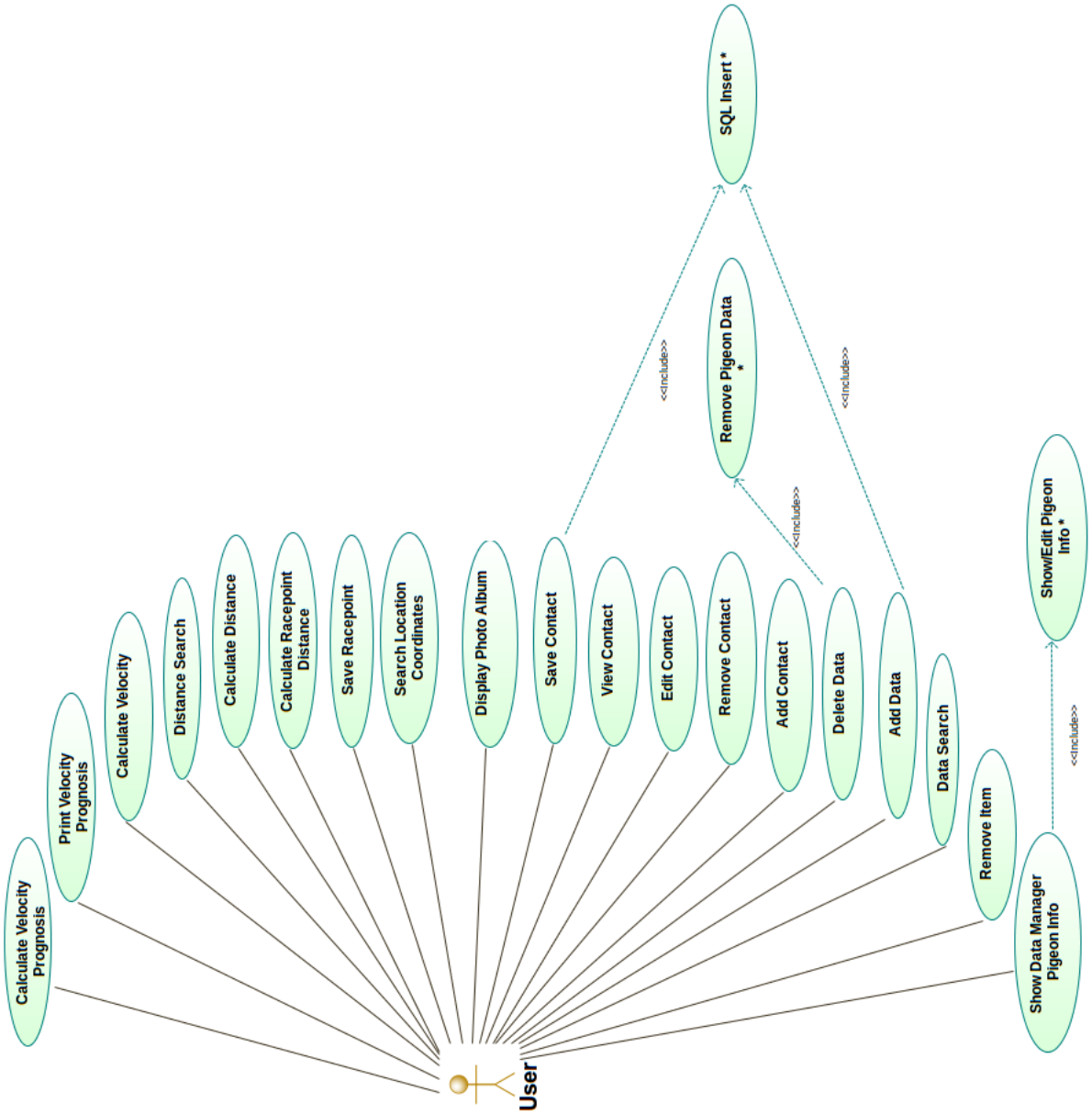

Figura C.17: Modelo obtenido con SAPSI correspondiente a las funcionalidades de las herramientas provistas por el sistema Pigeon Planner.

Pestañas

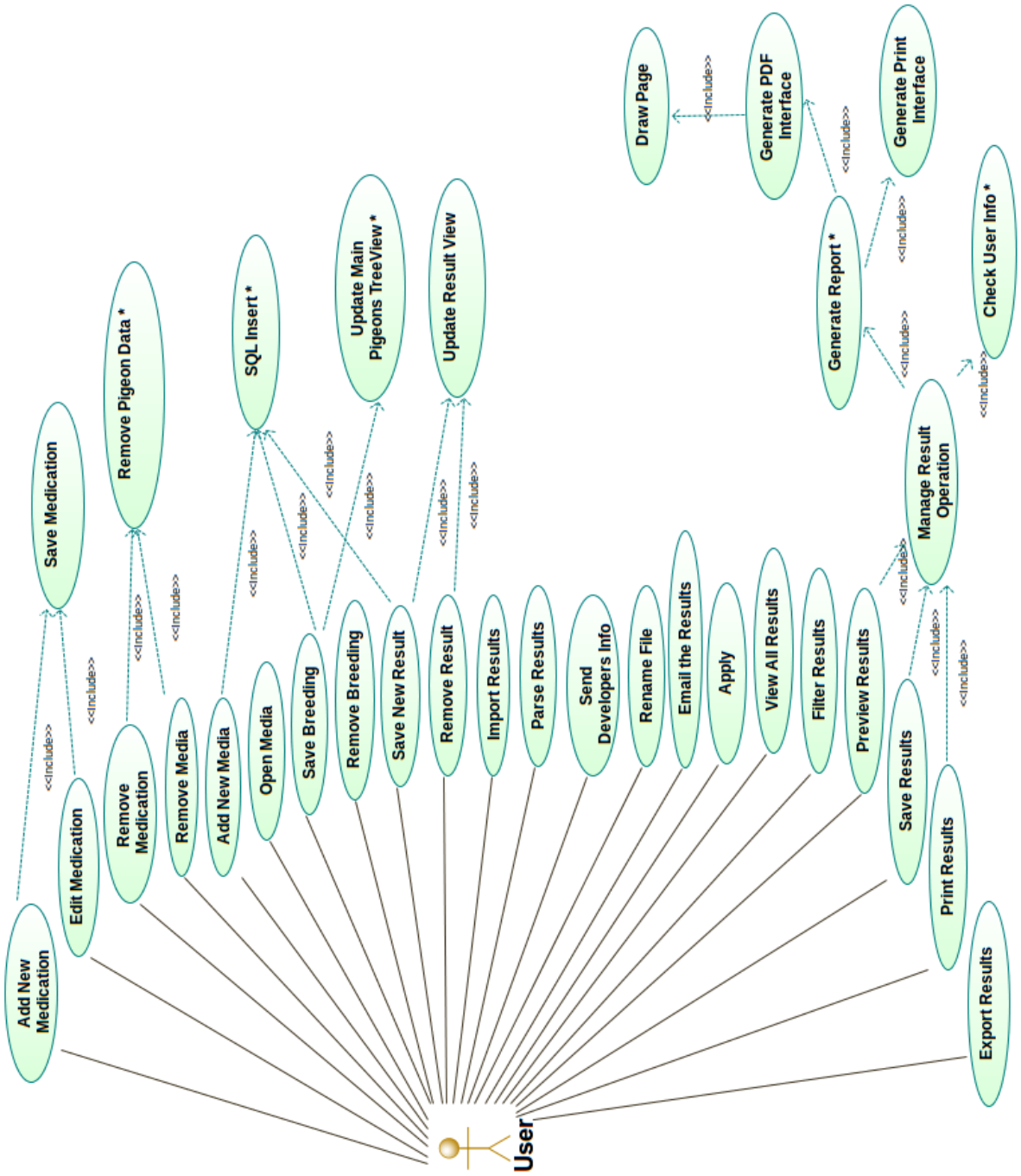

Figura C.18: Modelo obtenido con SAPSI de correspondiente a las funcionalidades encontradas en las pestañas del sistema Pigeon Planner.

Anexo D

Código Fuente y Archivo Glade del Ejemplo Libreta de Direcciones

No vale la pena aprender un lenguaje que no afecte la forma en que piensas acerca de la programación.
—Alan Perlis

En este anexo se muestra el código fuente del ejemplo de la aplicación que implementa una libreta de direcciones sencilla. Además se presentan el contenido del archivo glade para la misma aplicación, el cual fue generado con un constructor de interfaces gráficas.

Código Python

```
#!/usr/bin/env python
# encoding: utf-8

import sys
import os

try:
  import pygtk
  import gtk
  import gtk.glade
except:
  print "GTK and PyGTK not installed."
  sys.exit(1)
try:
    from contact import Contact
```

```
except:
    pass

class Contact:
    def __init__(self, id, name, address="", phone=""):
        self.id=id
        self.name=name
        self.address=address
        self.phone=phone

    # setters
    def set_id(self,id):
        self.id=id

    def set_name(self,name):
        self.name=name

    def set_address(self,address):
        self.address=address

    def set_phone(self,phone):
        self.phone=phone

    # getters
    def get_id(self):
        return self.id

    def get_name(self):
        return self.name

    def get_address(self):
        return self.address

    def get_phone(self):
        return self.phone

    def __eq__(self, other):
        return(other!=None and isinstance(other, Contact) and
            self.name == other.get_name())

class AddressBookGUI:
    last_contact_id=0
    def __init__ (self):
        """This initializes some variables."""
        self.glade_file_name ="address_book.glade"

        self.builder = gtk.Builder()
        self.builder.add_from_file(self.glade_file_name)

        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        back_button.set_sensitive(False)
        forward_button.set_sensitive(False)
```

```
        dic = {
        "search_button_clicked" : self.search,
        "back_button_clicked" : self.back,
        "forward_button_clicked" : self.forward,
        "delete_button_clicked" : self.delete,
        "save_button_clicked" : self.save,
        "clear_button_clicked" : self.clear,
        "close_button_clicked" : self.close,
        "destroy" : gtk.main_quit
        }

        self.builder.connect_signals(dic)

        self.contacts={}

        # used to naviagate using >> and << buttons
        self.record_number=0
        self.searched_list=[]
        self._load_contacts()
        self.main_window= self.builder.get_object("main_window").show_all()
#-----------------------------------------------
    def _load_contacts(self):
        try:
            file=open("Data.txt","r")
        except IOError:
            self.info_dialog("Error opening a file")
        line=file.readline()
        line=line.strip('\n')

        while line != "":
            if(line.find('*')!=-1):
                #self.last_contact_id= int(line[1:])
                last_contact_id= int(line[1:])
            else:
                id= int(line)
                name=file.readline().strip('\n')
                address=file.readline().strip('\n')
                phone=file.readline().strip('\n')
                contact = Contact(id, name, address, phone)
                self.contacts[name]=contact
            line=file.readline().strip('\n')

        file.close()
#-----------------------------------------------
    def search (self,widget):
        """get term to search from name text field"""

        name_entry = self.builder.get_object("name_entry")
        address_entry = self.builder.get_object("address_entry")
        phone_entry = self.builder.get_object("phone_entry")
        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        aux_name= name_entry.get_text();
```

```
        # clear contents of arraylist
        self.searched_list=[]

        # intialize recordNumber to zero
        self.record_number = 0

        if (aux_name==""):
            self.info_dialog("Please enter contact name to search")
        else:
            #get an array list of searched contacts
            self.searched_list = self.search_similar_contacts(aux_name)


            if(self.searched_list==[]):
                self.info_dialog("No record found.")
                back_button.set_sensitive(False)
                forward_button.set_sensitive(False)
            else:
                #downcast the object from list to Contact*/
                contact = self.searched_list[self.record_number];

                #displaying search record in text fields
                name_entry.set_text(contact.get_name())
                address_entry.set_text(contact.get_address())
                phone_entry.set_text(contact.get_phone())

                forward_button.set_sensitive(True)
                back_button.set_sensitive(True)
#------------------------------------------------
    def search_similar_contacts(self,name):
        search_result=[]
        for i in self.contacts:
            contact=self.contacts[i]
            if(contact.get_name().find(name)!=-1):
                search_result.append(contact)
        return search_result
#------------------------------------------------
    def contains_contact(self,name):
        search_result=False
        for i in self.contacts:
            contact=self.contacts[i]
            if(contact.get_name()==name):
                search_result=True
        return search_result
#------------------------------------------------
    def back (self,widget):
        # dec in recordNumber to display previous contact info
        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        name_entry = self.builder.get_object("name_entry")
        address_entry = self.builder.get_object("address_entry")
        phone_entry = self.builder.get_object("phone_entry")
```

```
        self.record_number-=1
        if(self.record_number<0):
            self.info_dialog("You have reached at begining of results")

            # if user has reached the begining, disable back button
            forward_button.set_sensitive(True)
            back_button.set_sensitive(False)
            self.record_number+=1
        else:
            forward_button.set_sensitive(True)
            contact = self.searched_list[self.record_number]

            name_entry.set_text(contact.get_name())
            address_entry.set_text(contact.get_address())
            phone_entry.set_text(contact.get_phone())
#-----------------------------------------------
    def forward (self,widget):
        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        name_entry = self.builder.get_object("name_entry")
        address_entry = self.builder.get_object("address_entry")
        phone_entry = self.builder.get_object("phone_entry")

        # inc in recordNumber to display next contact info
        self.record_number+=1
        if(self.record_number >= len(self.searched_list)):
            self.info_dialog("You have reached at end of search results")
            #if user has reached the end, disable froward button
            forward_button.set_sensitive(False)
            back_button.set_sensitive(True)
            self.record_number-=1
        else:
            back_button.set_sensitive(True)
            contact = self.searched_list[self.record_number]
            name_entry.set_text(contact.get_name())
            address_entry.set_text(contact.get_address())
            phone_entry.set_text(contact.get_phone())

#-----------------------------------------------
    def delete (self,widget):
        # used to delete a contac from the address book.
        name_entry = self.builder.get_object("name_entry")
        aux_name = name_entry.get_text()

        if(aux_name==""):
             self.info_dialog("Please enter contact name to delete.")
        else:
            #remove Contact of the given name
            contains_name = self.search_similar_contacts(aux_name)

            if(contains_name!=[]):
                for c in contains_name:
                    #checks if the contact is the user wants to delete
                    if(c.get_name()==aux_name):
```

```
                        del contains_name[c.get_name()]
                        del c
                        self.info_dialog("Contact deleted.")
            else:
                self.info_dialog("The contact does not exist.")

#-----------------------------------------------
    def save (self, widget):
        # used to save a contact in the address book
        # get values from text fields
        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        name_entry = self.builder.get_object("name_entry")
        address_entry = self.builder.get_object("address_entry")
        phone_entry = self.builder.get_object("phone_entry")

        aux_name = name_entry.get_text()
        aux_address = address_entry.get_text()
        aux_phone = phone_entry.get_text()

        if (aux_name==""):
            self.info_dialog("Please enter contact name.")
        else:
            if (not self.contains_contact(aux_name)):
                #create a new contact object and save it
                self.last_contact_id+=1
                contact = Contact(self.last_contact_id,\
                 aux_name, aux_address, aux_phone)
                self.contacts[aux_name]=contact
                self.info_dialog("Record added.")
            else:
                self.info_dialog("Record already inserted.")
#-----------------------------------------------
    def clear(self,widget):
        #clear text fields
        back_button = self.builder.get_object("back_button")
        forward_button = self.builder.get_object("forward_button")
        name_entry = self.builder.get_object("name_entry")
        address_entry = self.builder.get_object("address_entry")
        phone_entry = self.builder.get_object("phone_entry")

        name_entry.set_text("")
        address_entry.set_text("")
        phone_entry.set_text("")

        #clear contents of search arraylist
        self.record_number = -1;
        self.searched_list=[]
        forward_button.set_sensitive(False)
        back_button.set_sensitive(False)
#-----------------------------------------------
    def close(self,widget):
        try:
            file = open("Data.txt","w")
```

```
        except IOError:
            self.info_dialog("Error creating the file")
        file.write("*" + str(self.last_contact_id)+"\n")
        for c in self.contacts:
            contact=self.contacts[c]
            file.write(str(contact.get_id())+"\n")
            file.write(str(contact.get_name())+"\n")
            file.write(str(contact.get_address())+"\n")
            file.write(str(contact.get_phone())+"\n")
        file.close()
        gtk.main_quit()
#-----------------------------------------------
    def info_dialog(self, text=''):
        info_dialog = self.builder.get_object('info_dialog')
        info_dialog.set_transient_for(self.main_window)
        if (text != ''):
            info_dialog.set_markup(' '+text+' ')
        answer = info_dialog.run()
        info_dialog.hide()
#-----------------------------------------------
if __name__ == "__main__":
    instance = AddressBookGUI()
    gtk.main()
```

Archivo del constructor de GUI (Glade)

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <requires lib="gtk+" version="2.24"/>
  <!-- interface-naming-policy project-wide -->
  <object class="GtkMessageDialog" id="info_dialog">
    <property name="can_focus">False</property>
    <property name="border_width">5</property>
    <property name="type_hint">dialog</property>
    <property name="skip_taskbar_hint">True</property>
    <property name="buttons">close</property>
    <child internal-child="vbox">
      <object class="GtkVBox" id="info_dialog_vbox">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="spacing">2</property>
        <child internal-child="action_area">
          <object class="GtkHButtonBox" id="dialog-action_area">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="layout_style">end</property>
            <child>
              <placeholder/>
            </child>
            <child>
              <placeholder/>
            </child>
          </object>
          <packing>
            <property name="expand">True</property>
            <property name="fill">True</property>
            <property name="position">0</property>
```

```
        </packing>
      </child>
    </object>
  </child>
</object>
<object class="GtkWindow" id="main_window">
  <property name="can_focus">False</property>
  <signal name="destroy" handler="destroy" swapped="no"/>
  <child>
    <object class="GtkVBox" id="main_vbox">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <child>
        <object class="GtkTable" id="table_center_form">
          <property name="visible">True</property>
          <property name="can_focus">False</property>
          <property name="n_rows">3</property>
          <property name="n_columns">2</property>
          <child>
            <object class="GtkLabel" id="name_label">
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="xalign">0.20000000298023224</property>
              <property name="label" translatable="yes">Name</property>
              <attributes>
                <attribute name="foreground" value="#0f1005950595"/>
              </attributes>
            </object>
          </child>
          <child>
            <object class="GtkLabel" id="address_label">
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="xalign">0.20000000298023224</property>
              <property name="label" translatable="yes">Address</property>
              <attributes>
                <attribute name="foreground" value="#0f1005950595"/>
              </attributes>
            </object>
            <packing>
              <property name="top_attach">1</property>
              <property name="bottom_attach">2</property>
            </packing>
          </child>
          <child>
            <object class="GtkLabel" id="phone_label">
              <property name="visible">True</property>
              <property name="can_focus">False</property>
              <property name="xalign">0.20000000298023224</property>
              <property name="label" translatable="yes">Phone</property>
              <attributes>
                <attribute name="foreground" value="#17a009680968"/>
              </attributes>
            </object>
            <packing>
              <property name="top_attach">2</property>
              <property name="bottom_attach">3</property>
            </packing>
          </child>
          <child>
            <object class="GtkEntry" id="name_entry">
```

```
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="invisible_char">?</property>
            <property name="primary_icon_activatable">False</property>
            <property name="secondary_icon_activatable">False</property>
            <property name="primary_icon_sensitive">True</property>
            <property name="secondary_icon_sensitive">True</property>
          </object>
          <packing>
            <property name="left_attach">1</property>
            <property name="right_attach">2</property>
          </packing>
        </child>
        <child>
          <object class="GtkEntry" id="address_entry">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="invisible_char">?</property>
            <property name="primary_icon_activatable">False</property>
            <property name="secondary_icon_activatable">False</property>
            <property name="primary_icon_sensitive">True</property>
            <property name="secondary_icon_sensitive">True</property>
          </object>
          <packing>
            <property name="left_attach">1</property>
            <property name="right_attach">2</property>
            <property name="top_attach">1</property>
            <property name="bottom_attach">2</property>
          </packing>
        </child>
        <child>
          <object class="GtkEntry" id="phone_entry">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="invisible_char">?</property>
            <property name="primary_icon_activatable">False</property>
            <property name="secondary_icon_activatable">False</property>
            <property name="primary_icon_sensitive">True</property>
            <property name="secondary_icon_sensitive">True</property>
          </object>
          <packing>
            <property name="left_attach">1</property>
            <property name="right_attach">2</property>
            <property name="top_attach">2</property>
            <property name="bottom_attach">3</property>
          </packing>
        </child>
      </object>
      <packing>
        <property name="expand">True</property>
        <property name="fill">True</property>
        <property name="position">0</property>
      </packing>
    </child>
    <child>
      <object class="GtkToolbar" id="address_book_toolbar">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <child>
          <object class="GtkToolButton" id="search_button">
            <property name="visible">True</property>
```

```
      <property name="can_focus">False</property>
      <property name="tooltip_text" translatable="yes">Search
       contact by name </property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">search</property>
      <property name="use_underline">True</property>
      <property name="stock_id">gtk-find</property>
      <signal name="clicked" handler="search_button_clicked" swapped="no"/>
    </object>
    <packing>
      <property name="expand">False</property>
      <property name="homogeneous">True</property>
    </packing>
  </child>
  <child>
    <object class="GtkToolButton" id="back_button">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="tooltip_text" translatable="yes">Previous
       contact</property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">back</property>
      <property name="use_underline">True</property>
      <property name="stock_id">gtk-media-rewind</property>
      <signal name="clicked" handler="back_button_clicked" swapped="no"/>
    </object>
    <packing>
      <property name="expand">False</property>
      <property name="homogeneous">True</property>
    </packing>
  </child>
  <child>
    <object class="GtkToolButton" id="forward_button">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="tooltip_text" translatable="yes">Next
      contact</property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">forward</property>
      <property name="use_underline">True</property>
      <property name="stock_id">gtk-media-forward</property>
      <signal name="clicked" handler="forward_button_clicked" swapped="no"/>
    </object>
    <packing>
      <property name="expand">False</property>
      <property name="homogeneous">True</property>
    </packing>
  </child>
  <child>
    <object class="GtkToolButton" id="clear_button">
      <property name="visible">True</property>
      <property name="can_focus">False</property>
      <property name="tooltip_text" translatable="yes">Clean
      fields</property>
      <property name="use_action_appearance">False</property>
      <property name="label" translatable="yes">clear</property>
      <property name="use_underline">True</property>
      <property name="stock_id">gtk-clear</property>
      <signal name="clicked" handler="clear_button_clicked" swapped="no"/>
    </object>
    <packing>
```

```
            <property name="expand">False</property>
            <property name="homogeneous">True</property>
          </packing>
        </child>
        <child>
          <object class="GtkToolButton" id="delete_button">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="tooltip_text" translatable="yes">Delete
            visible contact</property>
            <property name="use_action_appearance">False</property>
            <property name="label" translatable="yes">delete</property>
            <property name="use_underline">True</property>
            <property name="stock_id">gtk-delete</property>
            <signal name="clicked" handler="delete_button_clicked" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="homogeneous">True</property>
          </packing>
        </child>
        <child>
          <object class="GtkToolButton" id="save_button">
            <property name="visible">True</property>
            <property name="can_focus">False</property>
            <property name="tooltip_text" translatable="yes">Save
            visible contact</property>
            <property name="use_action_appearance">False</property>
            <property name="label" translatable="yes">toolbutton</property>
            <property name="use_underline">True</property>
            <property name="stock_id">gtk-floppy</property>
            <signal name="clicked" handler="save_button_clicked" swapped="no"/>
          </object>
          <packing>
            <property name="expand">False</property>
            <property name="homogeneous">True</property>
          </packing>
        </child>
      </object>
      <packing>
        <property name="expand">False</property>
        <property name="fill">True</property>
        <property name="position">1</property>
      </packing>
    </child>
    <child>
      <object class="GtkTable" id="table_south_button">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="n_columns">6</property>
        <child>
          <object class="GtkButton" id="close_button">
            <property name="label" translatable="yes">Save &amp;amp; Close</property>
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="tooltip_text" translatable="yes">Save
            Address Book and Exit</property>
            <property name="use_action_appearance">False</property>
            <signal name="clicked" handler="close_button_clicked" swapped="no"/>
          </object>
```

```
            <packing>
              <property name="left_attach">5</property>
              <property name="right_attach">6</property>
            </packing>
          </child>
          <child>
            <placeholder/>
          </child>
          <child>
            <placeholder/>
          </child>
          <child>
            <placeholder/>
          </child>
          <child>
            <placeholder/>
          </child>
          <child>
            <placeholder/>
          </child>
          <child>
            <placeholder/>
          </child>
        </object>
        <packing>
          <property name="expand">True</property>
          <property name="fill">True</property>
          <property name="position">2</property>
        </packing>
      </child>
    </object>
  </child>
 </object>
</interface>
```

Referencias

Aguezzoul, A., Rabenasolo, B., y Jolly-Desodt, A.-M. (2006, Oct). Multicriteria Decision Aid Tool for Third-Party Logistics Providers' Selection. En *International conference on service systems and service management* (Vol. 2, p. 912-916). doi: 10.1109/ICSSSM.2006.320753

Ahamad, S., y Verma, G. N. (2016). Legacy program estimation. *International Journal of Computer Science and Information Security*, *14*(2), 30.

Aho, A., Hopcroft, J., y Ullman, J. (1988). *Estructuras de Datos y Algoritmos*. Addison-Wesley Iberoamericana.

Aho, A. V., Sethi, R., y Ullman, J. D. (2007). *Compilers: principles, techniques, and tools* (Vol. 2). Addison-wesley Reading.

Ali, K., y Lhoták, O. (2012). Application-only call graph construction. En J. Noble (Ed.), *Ecoop 2012 – object-oriented programming: 26th european conference, beijing, china, june 11-16, 2012. proceedings* (pp. 688–712). Berlin, Heidelberg: Springer Berlin Heidelberg. Descargado de `http://dx.doi.org/10.1007/978-3-642-31057-7_30` doi: 10.1007/978-3-642-31057-7_30

Alma-Team. (2007). *epl.di.uminho.pt/~gepl/ALMA*.

Almendros-Jimenez, J., y Iribarne, L. (2005). Designing GUI Components from UML Use Cases. En *Engineering of computer-based systems, 2005. ecbs'05. 12th ieee international conference and workshops on the* (pp. 210–217). Descargado de `http://ieeexplore.ieee.org/document/1409919/?arnumber=1409919` doi: 10.1109/ECBS.2005.31

Anderson, P., y Zarins, M. (2005). The codesurfer software understanding platform. En *Proceedings of the 13th international workshop on program comprehension* (pp. 147–148). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dx.doi.org/10.1109/WPC.2005.37` doi: 10.1109/WPC.2005.37

Andritsos, P., y Tzerpos, V. (2005). Information-theoretic software clustering. *Software Engineering, IEEE Transactions on*, *31*(2), 150–165. doi: 10.1109/TSE.2005.25

Anma, F., Ando, T., Itoh, R., Konishi, T., y Itoh, Y. (2002). The Method to Visualize the Domain-Oriented-Explanation of Program's Behaviors. En *Computers in education, 2002. proceedings. international conference on* (pp. 910–911).

Anma, F., Konishi, T., y Itoh, Y. (2004). Construction and Evaluation of an Educational System that Explains the Domain-Oriented-Explanation of Program's Behaviors. *Proc. of ICCE 2004*, 839–845.

Anquetil, N., y Lethbridge, T. (1997). File clustering using naming conventions for legacy systems. En *Proceedings of the 1997 conference of the centre for advanced studies on collaborative research* (p. 2).

Anquetil, N., y Lethbridge, T. C. (1999). Experiments with Clustering as a Software Remodularization Method. En *Reverse engineering, 1999. proceedings. sixth working conference on* (pp. 235–255).

Arango, G. (1989). Domain analysis: From art form to engineering discipline. En *Acm sigsoft software engineering notes* (Vol. 14, pp. 152–159).

Austin, M., y Samadzadeh, M. (2005). Software comprehension/maintenance: An introductory course. En *Systems engineering, 2005. icseng 2005. 18th international conference on* (pp. 414–419).

Bach, M. J., y cols. (1986). *The design of the unix operating system* (Vol. 1). Prentice-Hall Englewood Cliffs, NJ.

Backhouse, R. (1979). *Syntax of Programming Languages: Theory and Practice*. Prentice-Hall.

Ball, T. (1999). The Concept of Dynamic Analysis. En *Software Engineering - ESEC/FSE99* (pp. 216–234).

Ball, T., y Eick, S. (1996, abril). Software Visualization in the Large. *IEEE Computer*, *29*(4), 33–43. Descargado de `http://dx.doi.org/10.1109/2.488299` doi: 10.1109/2.488299

Ballestero, E., y Romero, C. (1998). Multiple Criteria Decision Making: An Introduction. En *Multiple criteria decision making and its applications to economic problems* (pp. 1–10). Boston, MA: Springer US. Descargado de `http://dx.doi.org/10.1007/978-1-4757-2827-9_1` doi: 10.1007/978-1-4757-2827-9_1

Bassil, S., y Keller, R. K. (2001). Software visualization tools: Survey and analysis. En *Program comprehension, 2001. iwpc 2001. proceedings. 9th international workshop on* (pp. 7–17).

Bauer, M., y Trifu, M. (2004). Architecture-aware adaptive clustering of oo systems. En *Software maintenance and reengineering, 2004. csmr 2004. proceedings. eighth european conference on* (pp. 3–14).

Belady, L. A., y Evangelisti, C. J. (1981). System partitioning and its measure. *Journal of Systems and Software*, *2*(1), 23–29.

Belmonte, J., y Dugerdil, P. (2010). Using Domain Ontologies in a Dynamic Analysis for Program Comprehension. En *Ontology-driven software engineering* (p. 8).

Bennett, K., y Rajlich, V. (2000). Software Maintenance and Evolution: a Roadmap. En *Proceedings of the conference on the future of software engineering* (pp. 73–87). New York, NY, USA: ACM.

Berón, M. (2010). Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Ph.D Thesis Dissertation at University of Minho. Braga. Portugal*.

Berón, M., Bernardis, H., Miranda, E., Riesco, D., Pereira, M. J., y Henriques, P. (2016). Measuring the understandability of wsdl specifications, web service understan-

ding degree approach and system. *Computer Science and Information Systems*, *13*(3), 779–807.

Berón, M., Henriques, P., Varanda Pereira, M. J., y Uzal, R. (2007a). PICS un Sistema de Comprensión e Inspección de Programas. *Congreso Argentino de Ciencias de la Computacion (CACIC07)*.

Berón, M., Henriques, P., Varanda Pereira, M. J., y Uzal, R. (2007b). Static and Dynamic Strategies to Understand C Programs by Code Annotation. *European Joint Conference on Theory and Practice of Software - Satellite Event - Open Cert.*.

Berón, M. M., Pereira, M. J. V., Oliveira, N., y da Cruz, D. (2010, June). Svs, bors, svsi: Three strategies to relate problem and program domains. En *2010 ieee 18th international conference on program comprehension* (p. 60-61). doi: 10.1109/ICPC.2010.24

Beyer, D., y Noack, A. (2005, May). Clustering software artifacts based on frequent common changes. En *13th international workshop on program comprehension (iwpc'05)* (p. 259-268). doi: 10.1109/WPC.2005.12

Biggerstaff, T. (1989, julio). Design Recovery for Maintenance and Reuse. *Computer*, *22*(7), 36–49.

Biggerstaff, T., Mitbander, B., y Webster, D. (1993). The Concept Assignment Problem in Program Understanding. En *Proceedings of the 15th international conference on software engineering* (pp. 482–498).

Binkley, D. (2007). Source Code Analysis: a Road Map. En *2007 Future of Software Engineering* (pp. 104–119).

Bojic, D., y Velasevic, D. (2000). A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering. En *Proceedings of the conference on software maintenance and reengineering* (p. 23-32). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dl.acm.org/citation.cfm?id=518900.795260`

Booch, G., James, R., y Jacobson, I. (2004). *The Unified Software Development Process.* Addison-Wesley Object Technology Series.

Botafogo, R. A., y Shneiderman, B. (1991). Identifying aggregates in hypertext structures. En *Proceedings of the third annual acm conference on hypertext* (pp. 63–74).

Bowman, I. T., y Holt, R. C. (1998). Software architecture recovery using Conway's law. En *Proceedings of the 1998 conference of the centre for advanced studies on collaborative research* (pp. 6–). IBM Press. Descargado de `http://dl.acm.org/citation.cfm?id=783160.783166`

Brandes, U. (2001). A Faster Algorithm for Betweenness Centrality*. *Journal of Mathematical Sociology*, *25*(2), 163–177.

Brans, J.-P., y Mareschal, B. (2005). PROMETHEE Methods. En *Multiple criteria decision analysis: State of the art surveys* (pp. 163–186). New York, NY: Springer New York. Descargado de `http://dx.doi.org/10.1007/0-387-23081-5_5` doi: 10.1007/0-387-23081-5_5

Brans, J.-P., Vincke, P., y Mareschal, B. (1986). How to Select and How to Rank Projects: the Promethee Method. *European Journal of Operational Research*, *24*(2), 228 - 238. Descargado de `http://www.sciencedirect.com/science/article/pii/0377221786900445` (Mathematical Programming Multiple Criteria Decision Making) doi: http://dx.doi.org/10.1016/0377-2217(86)90044-5

Brooks, R. (1983). Towards a Theory of the Comprehension of Computer Programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554.

Budd, T., Justice, T., y Pandey, R. (1995). General-Purpose Multiparadigm Programming Languages: an Enabling Technology for Constructing Complex Systems. En *Engineering of complex computer systems, 1995. held jointly with 5th csesaw, 3rd ieee rtaw and 20th ifac/ifip wrtp, proceedings., first ieee international conference on* (pp. 334–337).

Budgen, D., Burn, A. J., Brereton, O. P., Kitchenham, B. A., y Pretorius, R. (2011). Empirical evidence about the UML: a systematic literature review. *Software: Practice*

and Experience, *41*(4), 363–392. Descargado de `http://dx.doi.org/10.1002/spe.1009` doi: 10.1002/spe.1009

Carvalho, N. R., Almeida, J. J., Henriques, P. R., y Varanda, M. J. (2015). From source code identifiers to natural language terms. *Journal of Systems and Software*, *100*, 117–128.

Chidamber, S. R., y Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

Christen, P. (2008). Febrl: an Open Source Data Cleaning, Deduplication and Record Linkage System with a Graphical User Interface. En *Proceedings of the 14th acm sigkdd international conference on knowledge discovery and data mining* (pp. 1065–1068). doi: 10.1145/1401890.1402020

Christl, A., Koschke, R., y Storey, M.-A. (2007). Automated clustering to support the reflexion method. *Information and Software Technology*, *49*(3), 255 - 274. Descargado de `http://www.sciencedirect.com/science/article/pii/S095058490600187X` (12th Working Conference on Reverse Engineering) doi: http://dx.doi.org/10.1016/j.infsof.2006.10.015

Clarke, J., Dolado, J. J., Harman, M., Hierons, R., Jones, B., Lumkin, M., ... others (2003). Reformulating software engineering as a search problem. *IEE Proceedings-software*, *150*(3), 161–175.

Clayton, R., Rugaber, S., Taylor, L., y Wills, L. (1997). A Case Study of Domain-Based Program Understanding. En *Program comprehension, 1997. iwpc'97. proceedings., fifth iternational workshop on* (pp. 102–110).

Clayton, R., Rugaber, S., y Wills, L. (1998). Dowsing: a Tool Framework for Domain-Oriented Browsing of Software Artifacts. En *Automated software engineering, 1998. proceedings. 13th ieee international conference on* (pp. 204–207).

Coleman, D., Ash, D., Lowther, B., y Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44–49.

Cooper, A., Reimann, R., Cronin, D., y Noessel, C. (2014). *About Face: The Essentials of Interaction Design* (4th ed.). Wiley Publishing.

Corbi, T. (1989). Program Understanding: Challenge for the 1990s. *IBM Systems Journal*, *28*(2), 294 -306. doi: 10.1147/sj.282.0294

Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J., y van Deursen, A. (2007). Understanding Execution Traces Using Massive Sequence and Circular Bundle Views. En *Proceedings of the 15th ieee international conference on program comprehension* (pp. 49–58). Washington, DC, USA: IEEE Computer Society.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., y Koschke, R. (2009). A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, *35*(5), 684–702.

Couto, R., Ribeiro, A. N., y Campos, J. C. (2014, apr). Application of ontologies in identifying requirements patterns in use cases. *Electronic Proceedings in Theoretical Computer Science*, *147*, 62–76. doi: 10.4204/eptcs.147.5

Cross II, J., Barowski, L., Hendrix, D., y Teate, J. (1996). Control Structure Diagrams for Ada 95. En *Proceedings of the conference on tri-ada'96: Disciplined software development with ada* (pp. 143–147).

Cross II, J., Hendrix, D., y Maghsoodloo, S. (1998). The Control Structure Diagram: an Overview and Initial Evaluation. *Empirical Software Engineering*, *3*(2), 131–158.

Cuesta, A. G., y García, J. G. (2008). *Instrumentación de Código para el Cálculo de Tiempos de Ejecución y Respuesta en Sistemas de Tiempo Real* (Tesis de Master no publicada). Universidad de Cantabria. Santander.España.

De Lucia, A. (2001). Program Slicing: Methods and Applications. En *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on* (pp. 142–149).

Demeyer, S., Ducasse, S., y Lanza, M. (1999). A hybrid reverse engineering approach combining metrics and program visualisation. En *Reverse engineering, 1999. proceedings. sixth working conference on* (pp. 175–186).

De Silva, L., y Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151.

Ducasse, S., y Pollet, D. (2009). Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, *35*(4), 573–591.

Dugerdil, P., y Repond, J. (2010). Automatic Generation of Abstract Views for Legacy Software Comprehension. En *Proceedings of the 3rd india software engineering conference* (pp. 23–32).

Dugerdil, P., y Sennhauser, D. (2013). Dynamic decision tree for legacy use-case recovery. En *Proceedings of the 28th annual acm symposium on applied computing* (pp. 1284–1291). New York, NY, USA: ACM. Descargado de `http://doi.acm.org/10.1145/2480362.2480602` doi: 10.1145/2480362.2480602

Dujmović, J. (2013). Aggregation Operators and Observable Properties of Human Reasoning. En *Aggregation functions in theory and in practise* (pp. 5–16). Springer.

Dujmović, J., y Kadaster, M. (2002). A technique and tool for software evaluation. *Evolution*, *374*, 246.

Dujmović, J., Ralph, J., y Dorfman, L. (s.f.). Evaluation of Disease Severity and Patient Disability Using the LSP Method. En *Proceedings of the 12th information processing and management of uncertainty international conference (ipmu 2008)* (pp. 1398–1405).

Dujmović, J., y Tré, G. D. (2011). Multicriteria Methods and Logic Aggregation in Suitability Maps. *International Journal of Intelligent Systems*, *26*(10), 971–1001. Descargado de `http://dx.doi.org/10.1002/int.20509` doi: 10.1002/int.20509

Dujmovic, J. J. (2007, Dec). Continuous Preference Logic for System Evaluation. *IEEE Transactions on Fuzzy Systems*, *15*(6), 1082-1099. doi: 10.1109/TFUZZ.2007.902041

Dujmovic, J. J. (2008, June). Characteristic Forms of Generalized Conjunction/Disjunction. En *Fuzzy systems, 2008. fuzz-ieee 2008. (ieee world congress on computational intelligence). ieee international conference on* (p. 1075-1080). doi: 10.1109/FUZZY.2008.4630503

Dujmovic, J. j., y Elnicki, R. (1982). A DMS Cost/benefit Decision Model: Mathematical Models for Data Management System Evaluation, Comparison, and Selection. *National Bureau of Standards, Washington DC, No. GCR*, 82–374.

Dunsmore, A., Roper, M., y Wood, M. (2003, aug). The development and evaluation of three diverse techniques for object-oriented code inspection. *IEEE Transactions on Software Engineering*, *29*(8), 677–686. doi: 10.1109/tse.2003.1223643

Eastwood, A. (1993). Firm fires shots at legacy systems. *Computing Canada*, *19*(2), 17.

Eisenbarth, T., Koschke, R., y Simon, D. (2001). Aiding Program Comprehension by Static and Dynamic Feature Analysis. En *Proceedings of the ieee international conference on software maintenance (icsm'01)* (pp. 602–611). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dx.doi.org/10.1109/ICSM.2001.972777` doi: 10.1109/ICSM.2001.972777

Eisenbarth, T., Koschke, R., y Simon, D. (2003). Locating features in source code. *IEEE Transactions on software engineering*, *29*(3), 210–224.

Elish, M. O., y Al-Rahman Al-Khiaty, M. (2013). A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, *25*(5), 407–437. Descargado de `http://dx.doi.org/10.1002/smr.1549` doi: 10.1002/smr.1549

El-Ramly, M., Stroulia, E., y Sorenson, P. (2002). Mining System-User Interaction Traces for Use Case Models. En *Program comprehension, 2002. proceedings. 10th international workshop on* (p. 21-29). doi: 10.1109/WPC.2002.1021305

Erdemir, U., Tekin, U., y Buzluca, F. (2011). Object Oriented Software Clustering Based on Community Structure. En *Software engineering conference (apsec), 2011 18th asia pacific* (pp. 315–321).

Erdogmus, H., y Tanir, O. (Eds.). (2002). *Advances in Software Engineering*. New York, NY, USA: Springer-Verlag New York, Inc. doi: 10.1007/978-0-387-21599-0

Erlikh, L. (2000, May). Leveraging legacy system dollars for e-business. *IT Professional*, *2*(3), 17-23. doi: 10.1109/6294.846201

Escalona, M., Gutierrez, J. J., Mejías, M., Aragón, G., Ramos, I., Torres, J., y Domínguez, F. (2011). An overview on test generation from functional requirements. *Journal of Systems and Software*, *84*(8), 1379–1393.

Fonseca, R. F. C., Da Cruz, D. C., Henriques, P. M. R. S., y Pereira, M. J. V. (2008). How to interconnect operational and behavioral views of web applications. En *Program comprehension, 2008. icpc 2008. the 16th ieee international conference on* (pp. 263–267).

Fyson, J., y Boldyreff, C. (1998, marzo). Using Application Understanding to Support Impact Analysis. *Journal of Software Maintenance*, *10*(2), 93–110. Descargado de `http://dx.doi.org/10.1002/(SICI)1096-908X(199803/04)10:2<93::AID-SMR169>3.3.CO;2-Y` doi: 10.1002/(SICI)1096-908X(199803/04)10:2⟨93::AID-SMR169⟩3.3.CO;2-Y

Galde, U. I. D. (2016). *https://glade.gnome.org/*.

Garcia, J., Ivkovic, I., y Medvidovic, N. (2013). A comparative analysis of software architecture recovery techniques. En *Proceedings of the 28th ieee/acm international conference on automated software engineering* (pp. 486–496).

Gayo, J. E., Gil, J. M. M., Álvarez, A. M. F., y Chigne, H. (2003). A Generic E-Learning Multiparadigm Programming Language System: IDEFIX Project. En *Acm sigcse bulletin* (Vol. 35, pp. 391–395).

GEPL, U. d. M. (2017). *Language Specification and Processing Group (GEPL).* `http://www.di.uminho.pt/~gepl/`.

Graham, S., Kessler, P., y Mckusick, M. (1982). Gprof: a Call Graph Execution Profiler. *ACM Sigplan Notices*, *17*(6), 120–126.

Grove, D., y Chambers, C. (2001, noviembre). A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, *23*(6), 685–746. Descargado de `http://doi.acm.org/10.1145/506315.506316` doi: 10.1145/506315.506316

Gupta, R., Soffa, M. L., y Howard, J. (1997). Hybrid Slicing: Integrating Dynamic Information with Static Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(4), 370–397.

Haiduc, S., y Marcus, A. (2008). On the Use of Domain Terms in Source Code. En *Program comprehension, 2008. icpc 2008. the 16th ieee international conference on* (pp. 113–122).

Hamou-Lhadj, A., Braun, E., Amyot, D., y Lethbridge, T. (2005, March). Recovering behavioral design models from execution traces. En *Ninth european conference on software maintenance and reengineering* (p. 112-121). doi: 10.1109/CSMR.2005.46

Hamou-Lhadj, A., y Lethbridge, T. (2006). Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System. En *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on* (pp. 181–190).

Hamou-Lhadj, A., y Lethbridge, T. C. (2004). A survey of trace exploration tools and techniques. En *Proceedings of the 2004 conference of the centre for advanced studies on collaborative research* (pp. 42–55).

Helminen, J., y Malmi, L. (2010). Jype - a Program Visualization and Programming Exercise Tool for Python. En *Proceedings of the 5th international symposium on software visualization* (pp. 153–162).

Hendrix, D., Cross II, J., y Barowski, L. (2004). An Extensible Framework for Providing Dynamic Data Structure Visualizations in a Lightweight IDE. *ACM SIGCSE Bulletin*, *36*(1), 387–391.

Higo, Y., Kusumoto, S., y Inoue, K. (2008). A metric-based approach to identifying refactoring opportunities for merging code clones in a java software system. *Journal of Software Maintenance and Evolution: Research and Practice*, *20*(6), 435–461. Descargado de `http://dx.doi.org/10.1002/smr.394` doi: 10.1002/smr.394

Hutchens, D. H., y Basili, V. R. (1985). System structure analysis: Clustering with data bindings. *IEEE transactions on Software Engineering*(8), 749–757.

IEEE Standard for Software Maintenance. (1993). *IEEE Std 1219-1993*, 1-45. doi: 10.1109/IEEESTD.1993.115570

Ieee standard glossary of software engineering terminology. (1990). *IEEE Std 610.12-1990*, 1. doi: 10.1109/IEEESTD.1990.101064

Imagix-Corporation. (2016). *http://www.imagix.com/index.html*.

Jarke, M. (1999). Scenarios for modeling. *Communications of the ACM*, *42*(1), 47–48.

Jerding, D., y Stasko, J. (1998). The Information Mural: a Technique for Displaying and Navigating Large Information Spaces. *Visualization and Computer Graphics, IEEE Transactions on*, *4*(3), 257–271.

Johnson, S. C. (1975). *Yacc: Yet another compiler-compiler* (Vol. 32). Bell Laboratories Murray Hill, NJ.

Jungnickel, D. (2007). *Graphs, Networks and Algorithms* (3rd ed.). Springer Publishing Company, Incorporated.

JUNG-Team. (2016). *http://jung.sourceforge.net/*.

Kamran, M., Ali, M., y Akbar, B. (2016). Identification of core architecture classes for object-oriented software systems. *Journal of Applied Computer Science & Mathematics*, *10*(22).

Kamran, M., Azam, F., y Khanum, A. (2013). Discovering core architecture classes to assist initial program comprehension. En *Proceedings of the 2012 international conference on information technology and software engineering* (pp. 3–10).

Kasto, N., y Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. En *Proceedings of the fifteenth australasian computing education conference-volume 136* (pp. 59–65).

Keeney, R., y Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. J. Wiley, New York.

Kenneth, L. C. (1997). *Compiler Construction Principles and Practice*. PSW Publishing Company.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, *46*(5), 604–632.

Ko, A. J., DeLine, R., y Venolia, G. (2007). Information needs in collocated software development teams. En *Software engineering, 2007. icse 2007. 29th international conference on* (pp. 344–353).

Kobourov, S. G. (2012). Spring embedders and force directed graph drawing algorithms. *arXiv preprint arXiv:1201.3011*.

Korel, B., y Laski, J. (1988). Dynamic Program Slicing. *Information Processing Letters*, *29*(3), 155–163.

Koschke, R. (2002). Atomic architectural component recovery for program understanding and evolution. En *Software maintenance, 2002. proceedings. international conference on* (pp. 478–481).

Koschke, R., y Eisenbarth, T. (2000). A framework for experimental evaluation of clustering techniques. En *Proceedings iwpc 2000. 8th international workshop on program comprehension* (p. 201-210). doi: 10.1109/WPC.2000.852494

Krause, A. (2007). *Foundations of gtk+ development*. Apress.

Kuhn, A., Ducasse, S., y Girba, T. (2005, Nov). Enriching reverse engineering with semantic clustering. En *12th working conference on reverse engineering (wcre'05)* (p. 10 pp.-). doi: 10.1109/WCRE.2005.16

Lanza, M., Ducasse, S., Gall, H., y Pinzger, M. (2005). Codecrawler-an information visualization tool for program comprehension. En *Software engineering, 2005. icse 2005. proceedings. 27th international conference on* (pp. 672–673).

Lee, S.-W., Gandhi, R., Muthurajan, D., Yavagal, D., y Ahn, G.-J. (2006). Building problem domain ontology from security requirements in regulatory documents. En *Proceedings of the 2006 international workshop on software engineering for secure systems* (pp. 43–50).

Letovsky, S. (1986). Cognitive Processes in Program Comprehension. En *Papers presented at the first workshop on empirical studies of programmers* (pp. 58–79). Norwood, NJ, USA: Ablex Publishing Corp. Descargado de `http://dl.acm.org/citation.cfm?id=21842.28886`

Li, Q., Hu, S., Chen, P., Wu, L., y Chen, W. (2007, Aug). Discovering and Mining Use Case Model in Reverse Engineering. En *Fuzzy systems and knowledge discovery, 2007. fskd 2007. fourth international conference on* (Vol. 4, p. 431-436). doi: 10.1109/FSKD.2007.255

Lieberman, H., y Fry, C. (1995). Bridging the Gulf Between Code and Behavior in Programming. En *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 480–486).

Lientz, B. P., y Swanson, E. B. (1981, noviembre). Problems in application software maintenance. *Commun. ACM*, *24*(11), 763–769. Descargado de `http://doi.acm.org/10.1145/358790.358796` doi: 10.1145/358790.358796

Lindvall, M., y Sandahl, K. (1998). Traceability aspects of impact analysis in object-oriented systems. *Journal of Software: Evolution and Process*, *10*(1), 37–57.

Littman, D., Pinto, J., Letovsky, S., y Soloway, E. (1987). Mental Models and Software Maintenance. *Journal of Systems and Software*, *7*(4), 341–355.

Lutellier, T., Chollak, D., Garcia, J., Tan, L., Rayside, D., Medvidovic, N., y Kroeger, R. (2015). Comparing software architecture recovery techniques using accurate dependencies. En *Software engineering (icse), 2015 ieee/acm 37th ieee international conference on* (Vol. 2, pp. 69–78).

Lutz, R. (2001). Evolving good hierarchical decompositions of complex systems. *Journal of systems architecture*, *47*(7), 613–634.

Maalej, W., Tiarks, R., Roehm, T., y Koschke, R. (2014). On the comprehension of program comprehension. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *23*(4), 31.

Maletic, J. I., y Marcus, A. (2001). Supporting Program Comprehension Using Semantic and Structural Information. En *Proceedings of the 23rd international conference on software engineering* (pp. 103–112). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dl.acm.org/citation.cfm?id=381473.381484`

Mamaghani, A. S., y Meybodi, M. R. (2009). Clustering of software systems using new hybrid algorithms. En *Computer and information technology, 2009. cit'09. ninth ieee international conference on* (Vol. 1, pp. 20–25).

Mancoridis, S., Mitchell, B., Chen, Y., y Gansner, E. (1999). Bunch: a Clustering Tool for the Recovery and Maintenance of Software System Structures. En *Software maintenance, 1999.(icsm'99) proceedings. ieee international conference on* (pp. 50–59).

Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y.-F., y Gansner, E. R. (1998). Using Automatic Clustering to Produce High-Level System Organizations of Source Code. En *Iwpc* (Vol. 98, pp. 45–52).

Maqbool, O., y Babri, H. (2005). Interpreting clustering results through cluster labeling. En *Emerging technologies, 2005. proceedings of the ieee symposium on* (pp. 429–434).

Maqbool, O., y Babri, H. (2007). Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, *33*(11).

Maqbool, O., y Babri, H. A. (2004). The Weighted Combined Algorithm: a Linkage Algorithm for Software Clustering. En *Software maintenance and reengineering, 2004. csmr 2004. proceedings. eighth european conference on* (pp. 15–24).

Maqbool, O., y Babri, H. A. (2006). Automated software clustering: An insight using cluster labels. *Journal of Systems and Software*, *79*(11), 1632–1648.

Mathias, K. S., Cross II, J. H., Hendrix, T. D., y Barowski, L. A. (1999). The role of software measures and metrics in studies of program comprehension. En *Proceedings of the 37th annual southeast regional conference (cd-rom)* (p. 13).

McKee, J. R. (1984). Maintenance as a function of design. En *Proceedings of the july 9-12, 1984, national computer conference and exposition* (pp. 187–193). New York, NY, USA: ACM. Descargado de `http://doi.acm.org/10.1145/1499310.1499334` doi: 10.1145/1499310.1499334

Memon, A., Banerjee, I., y Nagarajan, A. (2003). GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. En *2013 20th working conference on reverse engineering (wcre)* (pp. 260–260).

Mens, K., Mens, T., y Wermelinger, M. (2002). Supporting Software Evolution with Intentional Software Views. En *Proceedings of the international workshop on principles of software evolution* (pp. 138–142).

Mernik, M., Lenič, M., Avdičaušević, E., y Žumer, V. (2002). Lisa: An interactive environment for programming language development. En *Compiler construction* (pp. 1–4).

Meyer, P., Siy, H., y Bhowmick, S. (2014). Identifying important classes of large software systems through k-core decomposition. *Advances in Complex Systems*, *17*(07n08), 1550004.

Miranda, E., Berón, M., Germán, M., Daniel, R., y Narayan, D. (2015). A Strategy for Detecting and Clustering Functionalities in Object Oriented Systems. En *Computers and their applications (cata-2015), isca 30th international conference* (Vol. 1). Honolulu, Hawaii, USA.

Miranda, E., Berón, M., Germán, M., Pereira, M. J. V., y Henriques, P. (2013). NESSy: a New Evaluator for Software Development Tools. En *2nd symposium on languages, applications and technologies* (Vol. 29, pp. 21–37). Dagstuhl, Germany.

Mitchell, B. S., y Mancoridis, S. (2001). Comparing the decompositions produced by software clustering algorithms using similarity measurements. En *Proceedings ieee international conference on software maintenance. icsm 2001* (p. 744-753). doi: 10 .1109/ICSM.2001.972795

Modelio. (2016). *https://www.modelio.org/*.

Montazer, G. A., Saremi, H. Q., y Ramezani, M. (2009, octubre). Design a New Mixed Expert Decision Aiding System Using Fuzzy ELECTRE III Method for Vendor Selection. *Expert Syst. Appl.*, *36*(8), 10837–10847. Descargado de `http://dx.doi.org/10.1016/j.eswa.2009.01.019` doi: 10.1016/j.eswa.2009.01.019

Mousseau, V., Slowinski, R., y Zielniewicz, P. (2000). A User-oriented Implementation of the ELECTRE-TRI Method Integrating Preference Elicitation Support. *Computers & Operations Research*, *27*(7–8), 757 - 777. Descargado de `http://www.sciencedirect.com/science/article/pii/S0305054899001173` doi: http://dx.doi.org/10.1016/S0305-0548(99)00117-3

Müller, H. A., Orgun, M. A., Tilley, S. R., y Uhl, J. S. (1993). A Reverse-engineering Approach to Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, *5*(4), 181–204.

Murphy, G. C., Kersten, M., y Findlater, L. (2006). How are java software developers using the elipse ide? *IEEE software*, *23*(4), 76–83.

Murphy, G. C., Notkin, D., Griswold, W. G., y Lan, E. S. (1998). An Empirical Study of Static Call Graph Extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *7*(2), 158–191.

Murphy, G. C., Notkin, D., y Sullivan, K. (1995). Software Reflexion Models: Bridging the Gap Between Source and High-level Models. *ACM SIGSOFT Software Engineering Notes*, *20*(4), 18–28.

Nuno, C., J., A. J., J., V. P. M., y Pedro, H. (2012). Probabilistic SynSet Based Concept Location. En A. Simões, R. Queirós, y D. da Cruz (Eds.), *1st symposium on languages, applications and technologies* (Vol. 21, pp. 239–253). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Oliveira, N. (2009). *Improving Program Comprehension Tools for Domain Specific Languages* (Tesis de Master no publicada). University of Minho, Braga, Portugal.

Olsina, L., y Rossi, G. (2002). Measuring Web Application Quality with WebQEM. *IEEE MultiMedia, 2002*, *09*(4), 20–29. doi: http://doi.ieeecomputersociety.org/10.1109/MMUL.2002.1041945

OMG, O. M. G. (2017).

Parnin, C., y Rugaber, S. (2012). Programmer Information Needs After Memory Failure. En *Program comprehension (icpc), 2012 ieee 20th international conference on* (pp. 123–132).

Parr, T. (2013). *The definitive antlr 4 reference*. Pragmatic Bookshelf.

PCVIA, G. (2016). *http://wiki.di.uminho.pt/twiki/bin/view/Research/PCVIA/*.

Pereira, C., Martinez, L., y Favre, L. (2011). Recovering Use Case Diagrams from Object Oriented Code: an MDA-based Approach. En *Information technology: New generations (itng), 2011 eighth international conference on* (pp. 737–742).

Perin, F., Renggli, L., y Ressia, J. (2010). Ranking software artifacts. En *4th workshop on famix and moose in reengineering (famoosr 2010)* (Vol. 120).

Pigoski, T. (1996). *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. New York, NY, USA: John Wiley & Sons, Inc.

Port, O. (1988). The software trap – automate or else. *Business Week*, *9*(3051), 142–154.

Pothen, A. (1997). Graph partitioning algorithms with applications to scientific computing. En *ICASE/LaRC interdisciplinary series in science and engineering* (pp. 323–368). Springer Netherlands. doi: 10.1007/978-94-011-5412-3_12

Poulin, J. S. (1994). Measuring software reusability. En *Software reuse: Advances in software reusability, 1994. proceedings., third international conference on* (pp. 126–138).

Power, J. F., y Malloy, B. A. (2004). A metrics suite for grammar-based software. *Journal of Software: Evolution and Process*, *16*(6), 405–426.

Prechelt, L. (2000). An empirical comparison of seven programming languages. *Computer*, *33*(10), 23–29.

Rajlich, V., y Wilde, N. (2002). The Role of Concepts in Program Comprehension. En *Program comprehension, 2002. proceedings. 10th international workshop on* (pp. 271–278).

Raza, A., Vogel, G., y Plödereder, E. (2006). Bauhaus-a tool suite for program analysis and reverse engineering. En *Ada-europe* (Vol. 4006, pp. 71–82).

Rilling, J., y Klemola, T. (2003). Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics. En *Program comprehension, 2003. 11th ieee international workshop on* (pp. 115–124).

Robillard, M. P. (2005). Automatic generation of suggestions for program investigation. En *Acm sigsoft software engineering notes* (Vol. 30, pp. 11–20).

Rohatgi, A., Hamou-Lhadj, A., y Rilling, J. (2008). An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. En *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on* (pp. 236–241).

Roy, B. (1971). Problems and Methods with Multiple Objective Functions. *Mathematical Programming*, *1*, 239-266. Descargado de `http://dx.doi.org/10.1007/BF01584088` doi: 10.1007/BF01584088

Rugaber, S. (1995). Program Comprehension. *Encyclopedia of Computer Science and Technology*, *35*(20), 341–368.

Rugaber, S. (2000). The Use of Domain Knowledge in Program Understanding. *Annals of Software Engineering*, *9*(1-2), 143–192.

Saaty, T. L. (1990, 05 de septiembre). How to Make a Decision: the Analytic Hierarchy Process. *European Journal of Operational Research*, *48*(1), 9–26. Descargado de `http://www.sciencedirect.com/science/article/pii/037722179090057I` doi: http://dx.doi.org/10.1016/0377-2217(90)90057-I

Saaty, T. L. (2008). Decision making with the analytic hierarchy process. *International Journal of Services Sciences*, *1*(1), 83–98.

Safyallah, H., y Sartipi, K. (2006). Dynamic analysis of software systems using execution pattern mining. En *Program comprehension, 2006. icpc 2006. 14th ieee international conference on* (pp. 84–88).

Salah, M., Mancoridis, S., Antoniol, G., y Penta, M. D. (2006, March). Scenario-Driven Dynamic Analysis for Comprehending Large Software Systems. En *Conference on software maintenance and reengineering (csmr'06)* (p. 10 pp.-80). doi: 10.1109/CSMR.2006.47

Sato, T., Shizuki, B., y Tanaka, J. (2008). Support for Understanding GUI Programs by Visualizing Execution Traces Synchronized with Screen Transitions. En *Program comprehension, 2008. icpc 2008. the 16th ieee international conference on* (pp. 272–275).

Schwanke, R. W. (1991). An intelligent tool for re-engineering software modularity. En *Software engineering, 1991. proceedings., 13th international conference on* (pp. 83–92).

Schwanke, R. W., y Platoff, M. A. (1993). Cross references are features. En *Machine learning: From theory to applications* (pp. 107–123). Springer.

Seffah, A., Djouab, R., y Antunes, H. (2001). Comparing and reconciling usability-centered and use case-driven requirements engineering processes. En *User interface conference, 2001. auic 2001. proceedings. second australasian* (pp. 132–139).

Shatnawi, R. (2015). Deriving Metrics Thresholds Using Log Transformation. *Journal of Software: Evolution and Process*, *27*(2), 95–113. (JSME-14-0025.R2) doi: 10.1002/smr.1702

Shawky, D. M., y Abd-El-Hafiz, S. K. (2016). Characterizing software development method using metrics. *Journal of Software: Evolution and Process*, *28*(2), 82–96. Descargado de `http://dx.doi.org/10.1002/smr.1766` (JSME-15-0099.R1) doi: 10.1002/smr.1766

Shneiderman, B. (1980). *Software Psychology: Human Factors in Computer and Information Systems (Winthrop computer systems series)*. Winthrop Publishers.

Shneiderman, B., y Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: a Model and Experimental Results. *International Journal of Computer & Information Sciences*, *8*(3), 219–238.

Shtern, M., y Tzerpos, V. (2007). Lossless comparison of nested software decompositions. En *Reverse engineering, 2007. wcre 2007. 14th working conference on* (pp. 249–258).

Shtern, M., y Tzerpos, V. (2011, sep). Evaluating software clustering using multiple simulated authoritative decompositions. En *2011 27th IEEE international conference on software maintenance (ICSM)*. IEEE. doi: 10.1109/icsm.2011.6080802

Shtern, M., y Tzerpos, V. (2012, enero). Clustering Methodologies for Software Engineering. *Advances in Software Engineering*, *2012*, 1-18. Descargado de `http://dx.doi.org/10.1155/2012/792024` doi: 10.1155/2012/792024

Si, H., Li, Y., Chen, B., y Fang, W. (2013). A Method of Use Case Oriented Semi-automatic Reverse Engineering. *Journal of Computational Information System*, *9*(5), 2093–2101.

Siff, M., y Reps, T. (1999). Identifying Modules via Concept Analysis. *Software Engineering, IEEE Transactions on*, *25*(6), 749–768.

Sinha, S., Harrold, M. J., y Rothermel, G. (1999). System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow. En *Proceedings of the 21st international conference on software engineering* (pp. 432–441). New York, NY, USA: ACM. Descargado de `http://doi.acm.org/10.1145/302405.302675` doi: 10.1145/302405.302675

Smit, M., Stroulia, E., y Wong, K. (2008). Use Case Redocumentation From GUI Event Traces. En *Software maintenance and reengineering, 2008. csmr 2008. 12th european conference on* (pp. 263–268).

Sobell, M. G., y Helmke, M. (2005). *A practical guide to linux commands, editors, and shell programming*. Prentice Hall Professional Technical Reference.

Şora, I. (2015). Helping program comprehension of large software systems by identifying their most important classes. En *International conference on evaluation of novel approaches to software engineering* (pp. 122–140).

Spärck Jones, K. (2007, noviembre). Automatic Summarising: The State of the Art. *Inf. Process. Manage.*, *43*(6), 1449–1481. Descargado de `http://dx.doi.org/10.1016/j.ipm.2007.03.009` doi: 10.1016/j.ipm.2007.03.009

Stasko, J., Domingue, J., Brown, M., y Price, B. (Eds.). (1998). *Software Visualization: Programming as a Multimedia Experience*. Cambridge, MA: MIT Press.

Steidl, D., Hummel, B., y Juergens, E. (2012). Using network analysis for recommendation of central software classes. En *Reverse engineering (wcre), 2012 19th working conference on* (pp. 93–102).

Storey, M.-A. (1998). *A Cognitive Framework for Describing and Evaluating Software Exploration Tools* (Tesis Doctoral no publicada). Simon Fraser University.

Storey, M.-A. (2005). Theories, Methods and Tools in Program Comprehension: Past, Present and Future. *Proceedings of the 13th International Workshop on Program Comprehension*, 181–191.

Storey, M.-A., Fracchia, D., y Müller, H. (1997). Cognitive Design Elements to Support the Construction of a Mental Model During Software Visualization. *Program Comprehension, 1997. IWPC'97. Proceedings., Fifth Iternational Workshop on*, 17–28.

Storey, M.-A., y Muller, H. (1995). Manipulating and Documenting Software Structures Using SHriMP Views. En *Software maintenance, 1995. proceedings., international conference on* (pp. 275–284).

Storey, M.-A., Wong, K., y Muller, H. (1997). How Do Program Understanding Tools Affect How Programmers Understand Programs. En *Proceedings of the fourth working conference on reverse engineering (wcre '97)* (pp. 12–). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dl.acm.org/citation.cfm?id=832304.836998`

Streekmann, N. (2011). Introduction. En *Clustering-based support for software architecture restructuring* (pp. 1–6). Wiesbaden: Vieweg+Teubner Verlag. Descargado de `https://doi.org/10.1007/978-3-8348-8675-0_1` doi: 10.1007/978-3-8348-8675-0_1

Su, S. Y., Dujmovic, J., Batory, D., Navathe, S., y Elnicki, R. (1987, septiembre). A Cost-benefit Decision Model: Analysis, Comparison and Selection of Data Management. *ACM Trans. Database Syst.*, *12*(3), 472–520. Descargado de `http://doi.acm.org/10.1145/27629.33403` doi: 10.1145/27629.33403

Thung, F., Lo, D., Osman, M. H., y Chaudron, M. R. V. (2014). Condensing Class Diagrams by Analyzing Design and Network Metrics Using Optimistic Classification. En *Proceedings of the 22nd international conference on program comprehension* (pp. 110–121). New York, NY, USA: ACM. Descargado de `http://doi.acm.org/10.1145/2597008.2597157` doi: 10.1145/2597008.2597157

Tonella, P., Ricca, F., Pianta, E., y Girardi, C. (2003). Using keyword extraction for web site clustering. En *Web site evolution, 2003. theme: Architecture. proceedings. fifth ieee international workshop on* (pp. 41–48).

Triantaphyllou, E. (2000). *Multi-criteria decision making methods: A comparative study*. Springer US. Descargado de `https://books.google.com.ar/books?id=tuPGe_ur-TYC`

Tzerpos, V., y Holt, R. C. (1999, Oct). MoJo: a distance metric for software clusterings. En *Sixth working conference on reverse engineering (cat. no.pr00303)* (p. 187-193). doi: 10.1109/WCRE.1999.806959

Tzerpos, V., y Holt, R. C. (2000). ACDC: An Algorithm for Comprehension-Driven Clustering. En *Proceedings of the seventh working conference on reverse engineering (wcre'00)* (pp. 258–). Washington, DC, USA: IEEE Computer Society. Descargado de `http://dl.acm.org/citation.cfm?id=832307.837118`

Varanda Pereira, M. J., y Henriques, P. (2001). Visualization/Animation of Programs Based on Abstract Representations and Formal Mappings. En *Human-centric computing languages and environments, 2001. proceedings ieee symposia on* (pp. 373–381).

Verwaest, T. (2007). *Object-oriented component detection for software understanding* (Tesis Doctoral no publicada). Vrije Universiteit Brussel.

von Laszewski, G. (1993). *A collection of graph partitioning algorithms* (Inf. Téc.). Technical Report, Syracuse University.

Von Mayrhauser, A., y Vans, M. (1995). Program Comprehension During Software Maintenance and Evolution. *Computer*, *28*(8), 44–55.

Vranić, V. (2002). Towards Multiparadigm Software Development. *Journal of Computing and Information Technology*, *10*(2), 133–147.

Wampler, D., Clark, T., Ford, N., y Goetz, B. (2010). Multiparadigm Programming in Industry: A Discussion with Neal Ford and Brian Goetz. *Software, IEEE*, *27*(5), 61–64.

Wani, M., y Gandhi, O. (1999). Development of maintainability index for mechanical systems. *Reliability Engineering & System Safety*, *65*(3), 259–270.

Weiser, M. D. (1979). Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. *Ann Arbor, MI*.

Wen, Z., y Tzerpos, V. (2004, June). An effectiveness measure for software clustering algorithms. En *Proceedings. 12th ieee international workshop on program comprehension, 2004.* (p. 194-203). doi: 10.1109/WPC.2004.1311061

Wiggerts, T. A. (1997). Using clustering algorithms in legacy systems remodularization. En *Reverse engineering, 1997. proceedings of the fourth working conference on* (pp. 33–43).

Wilde, N., y Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software: Evolution and Process*, *7*(1), 49–62.

WISHERT, D. (1969). Mode analysis: a generalization of nearest neighbour which reduces chaining effects (with discussion). *Numerical taxonomy*, 282–311.

Wong, W. E., Gokhale, S. S., y Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems and Software*, *54*(2), 87–98.

Wu, J., Hassan, A. E., y Holt, R. C. (2005, Sept). Comparison of Clustering Algorithms in the Context of Software Evolution. En *21st ieee international conference on software maintenance (icsm'05)* (p. 525-535). doi: 10.1109/ICSM.2005.31

Wu, J., y Storey, M.-A. (2000). A Multi-Perspective Software Visualization Environment. , 15.

Xiao, C., y Tzerpos, V. (2005). Software clustering based on dynamic dependencies. En *Software maintenance and reengineering, 2005. csmr 2005. ninth european conference on* (pp. 124–133).

Yin, M., Li, B., y Tao, C. (2010). Using cognitive easiness metric for program comprehension. En *Software engineering and data mining (sedm), 2010 2nd international conference on* (pp. 134–139).

Zaidman, A., Calders, T., Demeyer, S., y Paredaens, J. (2005). Applying webmining techniques to execution traces to support the program comprehension process. En *Software maintenance and reengineering, 2005. csmr 2005. ninth european conference on* (pp. 134–142).

Zaidman, A., y Demeyer, S. (2008). Automatic identification of key classes in a software system using webmining techniques. *Journal of Software: Evolution and Process*, *20*(6), 387–417.

Zhang, L., Qin, T., Zhou, Z., Hao, D., y Sun, J. (2006). Identifying use cases in source code. *Journal of Systems and Software*, *79*(11), 1588 - 1598. Descargado de `http://www.sciencedirect.com/science/article/pii/S0164121206000550` (Software Cybernetics) doi: http://dx.doi.org/10.1016/j.jss.2006.02.032

Zhang, Q., Qiu, D., Tian, Q., y Sun, L. (2010). Object-oriented software architecture recovery using a new hybrid clustering algorithm. En *Fuzzy systems and knowledge discovery (fskd), 2010 seventh international conference on* (Vol. 6, pp. 2546–2550).

Zhang, X., Gupta, R., y Zhang, Y. (2003). Precise Dynamic Slicing Algorithms. En *Software engineering, 2003. proceedings. 25th international conference on* (pp. 319–329).

Zhou, X., Qian, J., Chen, L., y Xu, B. (2009). Automatic identification of use cases from codes: A user's goal driven approach. *Wuhan University Journal of Natural Sciences*, *14*(5), 409–414.

Ziadi, T., da Silva, M. A. A., Hillah, L. M., y Ziane, M. (2011). A fully dynamic approach to the reverse engineering of uml sequence diagrams. En *Engineering of complex computer systems (iceccs), 2011 16th ieee international conference on* (pp. 107–116).

Zuse, H. (1993). Criteria for program comprehension derived from software complexity metrics. En *Program comprehension, 1993. proceedings., ieee second workshop on* (pp. 8–16).